MASTER IN COMPUTER SCIENCE

GRID resource information services for scheduling in DIET

Frauenkron, Peter

Award date:
2006

FUNDP
Institute of Computer Science

Rue Grandgagnage, 21
B-5000 Namur Belgium

Grid Resource Information Services for Scheduling in DIET

Dissertation presented for obtaining
the grade as
Master in Computer Science
by

Peter FRAUENKRON

Promotor: **Prof. Vincent ENGLEBERT**

Academic Year 2005-2006

Grid Resource Information Services for Scheduling in DIET

Abstract

The objective of grid computing is to confederate heterogeneous and distributed computer resources so as to aggregate the power. The scheduler is one element necessary for obtaining high performance of the system and it requires monitoring tools that provide the information crucial for the scheduler to place the request on the different servers. Actually, DIET offers the collection of information by FAST [QUINSON, 02]. For this dissertation, a manager for performance prediction tools, called CoRI and a basic measurement tool are developed. The principles of the grid, the scheduling and performance prediction will be introduced by the illustration of DIET. CoRI's design and implementation will be expounded and an overview of some other performance prediction tools available will be shown. Finally the results of a scheduler based on this information will be commented.

Keywords: grid computing, monitoring, performance prediction, resource information service, scheduling

Services d'information sur les ressources grilles pour l'ordonnancement dans DIET

Résumé

L'objectif d'une grille de calcul est l'agrégation des ressources d'ordinateurs hétérogènes et distribuées afin de réunir leur puissance de calcul. Des outils de monitoring fournissent les informations cruciales à l'ordonnanceur pour attribuer les tâches aux serveurs . DIET, un intergiciel de grille, utilise un ensemble d'informations via FAST [QUINSON, 02]. Pour ce mémoire un manager pour des outils de prédiction de performance et un simple outil d'observation ont été élaborés. Les principes d'une grille, d'ordonnancement, de prédiction de performance et de monitoring seront introduits par l'illustration de DIET. Leurs conceptions et leurs implémentations seront également exposées. D'autres outils de prédiction de performance seront abordés. Finalement, les résultats d'un ordonnanceur utilisant les informations seront commentés.

Mots-clés: calcul de grille, monitoring, prédiction de performance, service d'information de ressources, ordonnanceur.

Acknowledgements

Great thanks are in order for everyone who helped me with this dissertation, especially to Vincent Englebert, who wanted to guide this work at Namur, and to Eddy Caron, my supervisor at Lyon, to the whole team GRAAL, especially Eric Boix, Holly Dail, Alan Su, Raphaël Bolze and Yves Caniou who revealed for me the mysteries of the program DIET and accessory programs. Finally I wish to thank Piers Eyre-Walker and Holly Dail for correcting my grammar and spelling mistakes in this manuscript.

Contents

Glossary

CORBA Common Object Request Broker Architecture

CoRI The Collectors of Resource Information is a management system for resource information using FAST and CoRI-Easy. See Chapter 5 for more information.

CoRI-Easy A basic dynamic system providing availability metrics. See Chapter 5.

DIET The Distributed Interactive Engineering Toolbox can be seen as a middleware for grid application. See Chapter 2 for more information.

FAST The Fast Agent's System Timer [QUINSON, 02] provides application-specific performance forecast from SeD level.

Ganglia Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. [MASSIE and al., 04].

Globus The Globus Toolkit is an open source software toolkit used for building Grid systems and applications. It is being developed by the Globus Alliance and many others all over the world. [GLOBUS]

Grid A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [FOSTER and al, 98].

GridRPC "GridRPC is an easy-to-use API for Grids based on the established remote procedure call model" [GRIDRPC]. The GridRPC is an extension of the remote process call. The mechanism of RPC is applied to the field of grid computing.
The GridRPC API allows clients to access in a unique way to existing grid computing systems like NetSolve, Ninf or DIET. For more details about this work see the articles [NAKADA and al, 02] and [SEYMOUR and al, 04].

LA The Local Agent is the access point for the SeD and informs the MA of the presence of the SeD. See Chapter 2.

Load balancing "The assignment of equivalent amounts of work to processors which will execute concurrently." [BERMAN, 99]

LDAP The Lightweight Directory Access Protocol, see [HOWES and al., 99] for more information.

MA The Master Agent is one of the components of the grid architecture. Among other things, it is the access point for the client and schedules the clients' request. See Chapter 2.

Metric The term metrics refers to the criteria used to evaluate the performance of the system. See Section 4.1 for a more detailed explication.

MPI The Message Passing Interface "is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users" [MPI]

NWS The Network Weather Service [WOLSKI and al, 99] is a dynamic system that provides availability metrics from SeD and agent level.

Performance estimate "A collection of data pertaining to the capabilities of a particular server in the context of the particular client request." [DAIL and al., 06]

PSE Problem Solving Environments

RPC The Remote Procedure Call is a protocol to simplify the implementation of distributed applications. A program should be able to use a function of another program that turns on another machine without knowing the underlying network details. The RPC works with the server-client model.

Scheduling "The assignment of work to resources within a specified timeframe." [BERMAN, 99] The Chapter 3 explains in details this operation.

SeD The Server Daemon is the calculating machine. Chapter 2 explains this component.

Introduction

Grid computing is becoming more and more important for sharing power resources all over the world. High performance schedulers that manage the assignment of request to these resources are therefore needed. As a key component of the system, it is crucial that schedulers get the most detailed information about the system components. Grid resource information services are consequently deployed which will provide different types of performance measurements.

The DIET toolkit facilitates the building of grids by acting as middleware. The toolkit includes in particular job scheduling and a grid architecture, but there is no default resource information service. Although the FAST and NWS resource information services can be used, we will show that these will not always provide the information needed, and so, optimal scheduling will not attainable. DIET consequently needs another service that can provide this information. In the literature, we can find many information services, some of which are functionally equal to FAST or NWS, but each one with different goals and features. DIET's scheduling could also benefit from these services. We have therefore developed the CoRI system, that not only provides basic resource information, but also allows an extension to other current or future information services.

The first chapter introduces the grid philosophy and different possible approaches to the implementation. The second chapter explains the engineering toolbox DIET, and the third chapter explains the job of the scheduler, some typologies and the implementation in DIET. Chapter 4 introduces the performance evaluation and explains the different types of resource information services. It moreover explains the need for a new service by describing the requirements and problems of resource information services. Finally, Chapter five describes the design and implementation of the the new tool CoRI, that can satisfy these needs.

Chapter 1

Grid computing

Before explaining the grid, the following example shows the difficulties and limits of the ordinary calculating machine paradigm. In this dissertation we will use the POV-Ray software for facilitating the comprehension of the grid. The *Persistence of Vision Raytracer* (POV-Ray) [POVRAY] is a high-quality, totally free tool for creating stunning three-dimensional graphics. The principle of POV-Ray is to create an image by using a simple text file as information (the image description file). Information in the file (i.e. the position of the light source, the position and structure of the objects, etc.) allows the program to build an image of high resolution and often with amazing results (see Figure 1.1)[1].

The problem of the program is the computational power and time needed to build this high-

Figure 1.1: POV-Ray example by gerberc(2003)

resolution image. On the one hand, the time it takes to calculate this image on an ordinary computer can exceed some hours: the user has to wait and the computer is occupied at nearly 100

[1] `http://www.renderosity.com/viewed.ez?galleryid=568961&Start=49&Artist=gerberc&ByArtist=Yes`

percent during this calculation. On the other hand we will have perhaps other machines accessible via the network and these machines may be unoccupied at this time.
The question now is what can be done to speed up the process of computing the image? The answer is to use a grid computing system that provides the POV-Ray service. We will designate this solution as *POV-RayGrid*.

1.1 What is computational grid?

A grid can be compared with an electric network (as well called a grid) where resources will be applied. Not electricity will be provided to an important number of persons but services like computing power, memory or storage space by bundling the resources available in a network.

The term **resources** has to be understood in the largest meaning, examples are not only compute cycles, disk capacity, or network capacity, but also applications, data, sensors, or even humans. In the example of POV-RayGrid the resources could be:

- the software POV-Ray,
- the hardware resources of the machines like number of CPUs, amount of memory, the bandwidth between the client and the server,...
- also the image description file can be a resource. The file resides on the local machine, for this reason we don't have to request it from the client.

A first definition of a computational grid can be found in the book "The Grid: Blueprint for a New Computing Infrastructure" [FOSTER and al, 98].

Definition 1 *A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.*

Why talking about an **infrastructure**? A computational grid must work with a large-scale federation of resources. To achieve the necessary interconnections the federation requires significant hardware infrastructure. To control and monitor the resulting collection the federation requires software infrastructure.

It is fundamental that the service has to be **dependable**. A grid must assure the adequate use of resources for delivering services under established Quality of Service (QoS) requirements. It seems to be logical as users to require assurances for qualities like response time, throughput, availability, security.
The challenge is to work with a co-allocation of multiple resource types to meet complex user demands. In the process, the utility of the combined system should be significantly greater than the sum of its parts.
The user could ask a grid with a response time and a throughput that are predictable, sustained, and of high level.

The "five nines" are a good example to show how important reliability is. Some systems have to run with 99.999 percent availability. It means in average the system is down only 5 minutes during the year. For example a computational grid is supporting the real time treatment of patient's medical images. What would happen if this system is down during a complex brain operation? The user needs the assurances of predictable, sustained, and often high levels of performance from each piece of the federation. The degree of reliability always depends on the application, POV-RayGrid didn't need a 5 nine platform.

A second fundamental concern is the need for **consistency of service**. To allow fundamental issues such as authentication, authorization, resource discovery, and resource access a Grid is built from multi-purpose protocols and interfaces. It is important that these protocols and interfaces are standard and open. The standards should hide heterogeneity without hindering the high-performance execution and allowing scalability and pervasive access.

If **pervasive access** is guaranteed the user can be sure that the services are always available, even if the user's environment is dynamic and in which resource failure is commonplace. This does not mean that resources are everywhere or universally available but that the grid must tailor its behavior as to extract the maximum performance from the available resources.

Finally, an infrastructure must offer **inexpensive access**. A computational grid should offer attractive costs.

Other main characteristics of the grid are extracted from the articles [FOSTER, 02], [BOTE-LORENZO and al, 04], and [FOSTER and al, 01].

Large scale: a grid must be able to deal with a number of resources ranging from just a few to millions. This raises the very serious problem of avoiding potential performance degradation as the grid size increases. A good example of this grid property is DIET, because it allows scalability by using a special architecture. See Chapter 2.2 for detailed information of this particularity.

Geographical distribution: grid's resources may be located at distant places. For example the Grid'5000 project has actual nine sites throughout France [GRID'5000].

Heterogeneity: a grid hosts both software and hardware resources that can be very varied ranging from data, files, software components or programs to sensors, scientific instruments, display devices, personal digital organizers, computers, super-computers and networks.

Resource sharing: resources in a grid belong to many different organizations that allow other organizations (i.e. users) to access them. External resources can thus be used by applications, promoting efficiency and reducing costs. The sharing is not only file exchange but rather direct access to computers, data, and other resources.

Multiple administrations: each organization may establish different security and administra-

tive policies under which their owned resources can be accessed and used. As a result, the already challenging network security problem is complicated even more with the need of taking into account all different policies.

Resource coordination: The key concept is the ability to negotiate resource-sharing arrangements among a set of participating parties (providers and consumers) and then to use the resulting resource pool for a purpose. Resources in a grid are not subject to centralized control. But the sharing must be highly-controlled by addressing questions like security, policy, payment, or membership in order to provide aggregated computing capabilities. The resulting set of individuals and/or institutions defined by such sharing rules form what we call a *virtual organization* (VO).

In POV-RayGrid, the participating parties are the users who want create a new image, an agent receiving and distributing the job and at least one server that is able to build the image. To define the VO, the access to this trio is limited by a password for the three components.

Transparent access: a grid should be seen as a single virtual computer. For this characteristic the grid can be seen as a new type of operating system. The client is not aware of the complexity of the underlying system.

1.2 Exploiting the grid

But how can grid computing now be used to deliver faster the answer to a request? If a problem is too big to be treated as one, divide to conquer is the slogan to follow. Therefore problems with high demand of resource will be subdivided in small problems. Also problems with a low demand of resources can overwhelm a single machine. Constant dripping wears away the stone, and in our context this would mean that machine's capacity will be overstepped by thousands or millions of small requests per second. The grid dispatches the work to computers that have enough power to treat a part of the problem. The parts can be treated at the same time, in parallel.
In the example of POV-RayGrid, either the user or the system has to specify the dimension of each sub-image. The first sub-image could be the first 20 resolution lines of the image; the second could be the image between the 21^{th} and the 40^{th} line, etc. Instead of sending only one request, the client will send many smaller requests, and each request will indicate the lines to process. Once the machines have created their sub-images they will return them to the client. The only thing the client has to do is to assemble the sub-images.

The result of this segmentation is very easy to illustrate. On the Figure 1.2, the image is cut into sub-images, but the creation of this one might be much faster than on a single machines.

1.3 Approaches to build a grid

Now that the principles of grid systems are clear, the question is how to implement the grid? Different approaches for grid programming are mentioned in the article [FOSTER and al, 98] and

Figure 1.2: POV-RayGrid example after segmentation

some of the well-known models are cited here. The first approach consists of **adapting models** that have already proved successful in sequential or parallel environments:

1. Using a grid-enabled shared-memory system (i.e. Teamster-G [LIANG and al, 05]),
2. using a grid-enabled MPI (i.e. MPICH-G2 [KARONIS and al, 03]),
3. using a grid-enabled file system.

Newer models like service-oriented architecture (SOA) could response to other types of requirements. One example is the JLab's Lattice Portal [WATSON and al., 02] that uses the technology of Web Service. Another approach is to use **specific technologies** that are effective in distributed computing like Remote Procedure Call (RPC), or the object-oriented model based on techniques such as the Common Object Request Broker Architecture (CORBA). The software engineering advantage of these technologies is that their encapsulation properties facilitate the modular construction of programs and the reuse of existing components. Some efforts of these different attempts to provide a programming model and a corresponding system or a language appropriated for the grid have been collected and catalogued by the Advanced Programming Models Research Group of the Global Grid Forum [LEE and al, 01].

As we can see, there are multiple approaches for building a grid. After this short introduction of grid models, we can now better describe in detail the DIET toolbox that uses one of these approaches.

Chapter 2

DIET, Distributed Interactive Engineering Toolbox

DIET is a toolbox for developing Application Service Provider systems easily (Network Enabled Server systems) on Grid platforms. It is developed in the Laboratory of Computer Science in Parallelism by the GRAAL team at the *École Normale Supérieure* in Lyon, France. The following chapter is widely inspired by the DIET User's Manual guide [BOLZE and al., 06]

2.1 Introduction

DIET is based on the scheme and follows the GridRPC API defined within the Global Grid Forum [GRIDRPC]. This chapter will explain the basic architecture of DIET. To know more about DIET and his additional tools (FAST for performance evaluation [QUINSON, 02], LogMgr for monitoring, VizDIET for the visualization, GoDIET for the deployment,...), the reader can visit the DIET home page `http://graal.ens-lyon.fr/DIET`.

The new resource information service CoRI-Easy and the information service manager CoRI are specially built for the DIET toolbox. So it would be necessary to understand this software environment and hence we can provide DIET useful and appropriated new tools.

DIET can be seen as a middleware for grids, this means an abstraction level is added on top of existing grid technologies. It is no longer necessary to implement a complete architecture, it is sufficient to implement the server and client side (the endpoints) for offering the functionality. Due to the new abstraction level, it is necessary to explain the different actors working with DIET:

DIET developers A DIET developer is a contributor of the DIET software. This person adds a component to DIET to improve the utility of the toolbox DIET.

DIET service developers A DIET service developer uses the DIET API with the purpose to build the endpoints of his own grid services.

Service users A service user launches the server and/or the client of a service created by a DIET service developer.

Our example of POV-Ray now changes its concept. The old *POV-RayGrid* constitutes a complete client-agent-server grid hierarchy. However the service implemented for the use in DIET does not need the agent part and will thereby constitute only server and client. Hence this change of needed components for a grid enabled POV-Ray service; we will designate this new service as *POV-RayService*.

A **service** in DIET is placed on the server side of the architecture and will answer client's **requests** (also called **jobs** or **task**) which are translations of general **problems** of the user.

2.2 GridRPC Programming Model

The GridRPC approach [SEYMOUR and al, 04] can be used to build Problem Solving Environments (PSE) on computational Grids. It includes the definition of an API and defines a model to perform remote computation on servers. In such paradigm, the architecture consists of a scheme that is close to the RPC (Remote Procedure Call) model. The Figure 2.1 shows this architecture.

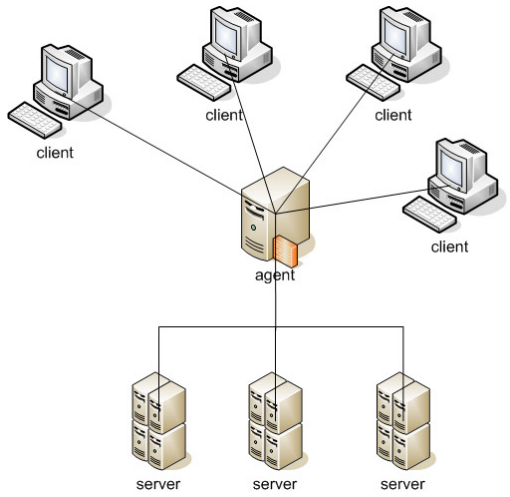

Figure 2.1: The GridRPC approach with clients, an agent and servers

Client The client wants to solve a problem. He is in possession of the description of the problem (the request). In the case of POV-RayGrid, the client would be the application that sends the request for computing the image by a given description file.

Server The server is the entity that solves the request. A server can be a simple home computer, a bash system or a mainframe.

Agent The agent receives the requests from the clients. He schedules these requests to the different SeDs he knows. This scheduling uses information about the performance of the platform gathered by an information service. The performance can include static and dynamic information such as software and hardware resources.

The GridRPC API defines the client API to send requests to a Network Enabled Server implementation. The function handle makes the link between the problem name on the client side and the instance of such service available on the server side. Other aspects defined in the GridRPC are the synchronous and asynchronous calls, a session ID for information about previous asynchronous requests and a wait function on the client side.

This API is instantiated by several middleware such as DIET, Ninf [NAKADA and al, 99], NetSolve [ARNOLD and al, 01], and XtremWeb [CAPPELLO and al., 05].

2.3 DIET components

At first, we have to know the architecture of DIET. Such one-agent-architectures are used in other grid middleware (i.e. NetSolve [ARNOLD and al, 01] or Ninf [NAKADA and al, 99]), but they would mean all the information has to go through one single point. And why is this a problem?

1. As we can see in Figure 2.1 this architecture would mean one *single point of failure*. If this agent fails, the whole system gives out.
   It is possible to fix this problem by using a backup agent that intervenes if the first agent falls out.

2. Secondary, the work load on the agent can slow down the whole system or even bring the system down.
   Let's imaging a single agent grid system for POV-RayService that consists of thousand of clients sending millions of description files to a thousand of servers. The time and capacity to coordinate all these requests would exceed the possibilities of one single machine.

Compared to this common architecture, the particularity of DIET is the avoidance of these disadvantages. Therefore the process of scheduling the requests is being modified. There is no single agent scheduling the requests, but DIET is distributing this job amongst a hierarchy of Local Agents (LA) and Master Agents (MA). With this approach a high performance and scalable environment can be obtained.

A brief description about the software architecture of DIET includes four different components. The server, called Server Daemon (SeD), contacts his local agent; the local agent contacts the master agent; and finally the client who contacts the master agent and the server daemon.

Figure 2.3 shows this architecture of DIET with the four main[1] components and their interactions.

[1]Other components are the CORBA nameservice, the NWS nameservice, the NWS memoryserver, and the NWS sensors. They are not presented in this Figure to facilitate the comprehension of the more important components

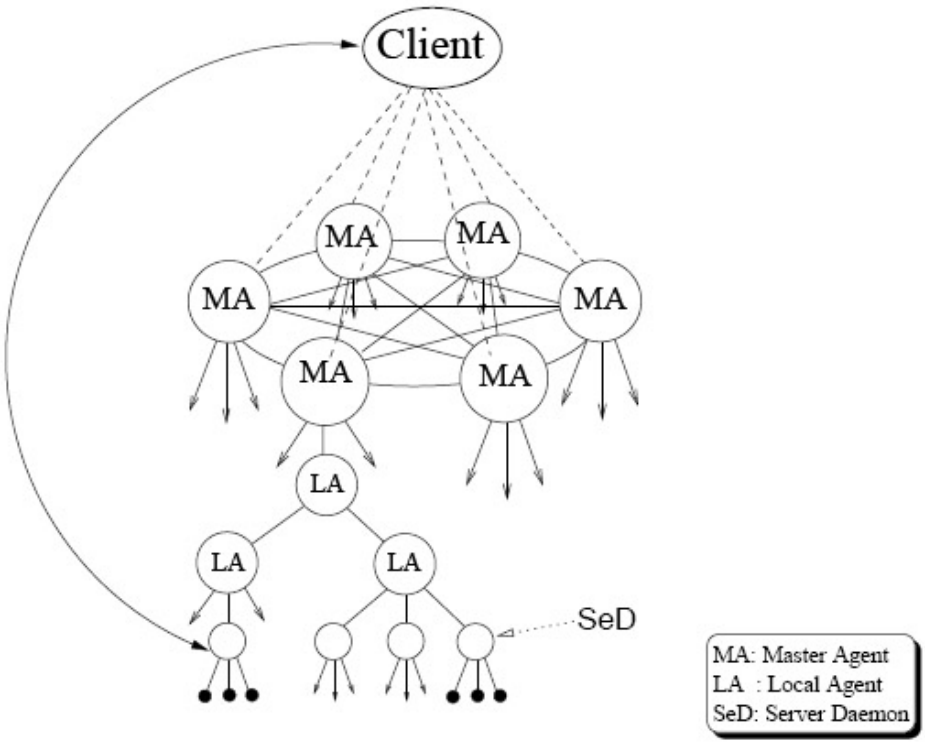

Figure 2.2: The global hierarchy of DIET's components, the clients, the master agents, the local agents and the servers [BOLZE and al., 06]

Client

A client is an application which uses DIET to solve problems. DIET provides multiple APIs for the client in order to connect to the master agent. The client can connect to DIET from a web page (via HTTP request), a problem solving environment (PSE) such as Matlab [2] or Scilab [3], or from a compiled program (by an interface written in c).

Master Agent

The MA is the contact point for the client because requests are sent from clients to MA. This agent is like a search engine, but instead of providing relevant internet pages for given

[2]Matlab is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. The reader can visit the home page of Matlab for more information: `www.mathworks.com/products/matlab/description1.html`

[3]Scilab Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications. The reader can visit the home page of Scilab for more information: `www.scilab.org`

keywords, it provides the client with addresses of appropriated servers for a given request.
A second role is to be a contact point for the local agent. So the master agent is contact point both for clients and for local agents.

But how does this connection to the MA work? The client has to know the contact address (there is no automatic detection implemented yet). So the client connects to an MA by a specific name server or by a web page which stores the various MA locations. Once the client is connected to the MA and once the client's request is received, the MA collects computation abilities from the servers and chooses the best one. The reference of the chosen server is returned to the client. Finally, due to the higher abstraction, the MA does not need any change for supporting whatever program that will run on the server.

Local Agent

Like a search engine, the MA needs information about servers from reliable sources. Search engines have tools like crawlers for receiving this information, and in DIET the local agents are providing this information. An LA is used as middleman between MA and servers. Therefore the LA performs a partial scheduling on its sub-tree, which reduces the load at the MA.

For achieving this goal, the LA stores a list of children (other LAs or/and servers), specifying for each child the services it is able to treat. For reducing a maximum the load at the MA and as well for the LAs, a hierarchy of LAs may be deployed between an MA and the servers.

Server Daemon

The Server Daemon (SeD) encapsulates a computational server and offers server's resources. It is responsible for computing requests arriving from the client. But it is not enough to offer only its resources, additionally it must give indications about its resources and their load for simplifying the activity of the LA and MA.

For providing the resources of the server, the SeD offers services by declaring these services to its parent LA or MA. For informing the MA and LA of its resources, the SeD stores information like data types available locally (i.e. on the server), a list of problems that can be solved on it, and performance-related information such as the amount of available memory or the number of processors available.
This performance-related information can be retrieved by modules like NWS or FAST. Both modules are described in Chapter 4.

In the POV-Ray example, two elements must be present at SeD level. The POV-Ray software must be installed, and the SeD must offer the service "POV-RayService" to their LA.

A special connection between master agents is possible, but only if special software is activated, either JXTA [CARON and al.] or Multi-MA [DAHAN, 05]. This connection has a different meaning compared to the father-son meaning of the LA-SeD or MA-LA connection.

2.4 DIET initialization

"Figure 2.3 shows each step of the initialization of a simple Grid system. The architecture is built in hierarchical order, each component connecting to its parent. The MA is the first entity to be started (1). It waits for connections from LAs or requests from clients.

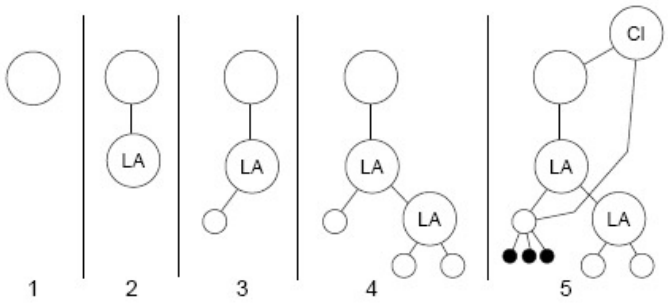

Figure 2.3: Initialization of a DIET system [BOLZE and al., 06].

In step (2), an LA is launched and registers itself with the MA. At this step of system initialization, two kinds of components can connect to the LA: a SeD (3), which manages some computational resources, or another LA (4), to add a hierarchical level in this branch. When the SeD registers to its parent LA, it submits a list of the services it offers. The agent then reports the new service offering through its parent agent until the MA. If the service was previously unavailable along that branch of the hierarchy the local and the master agents update their records. Finally, clients can access the registered service by contacting the MA (5) to get a reference to the best SeD available." [BOLZE and al., 06]

2.5 Solving a problem

"This section will present the algorithm of DIET for selecting a server for the computation among those available. We assume that the architecture described in Section 2.2 includes several servers able to solve the same problem. This decision is made in six steps.

- The MA propagates the client request through its sub-trees down to the capable servers; actually, the agents only forward the request on those sub-trees offering the service.
- Each addressed server will send its performance-related information for processing this request to its "parent" (an LA).
- Each LA that receives one or more positive responses from its children sorts the servers and forwards the best responses to the MA through the hierarchy.

- Once the MA has collected all the responses from its children, it chooses a pool of fast servers and sends their references to the client.
- When the client receives the reference of best SeD from his MA, he contacts the SeD directly for submitting the request.
- Once the computation is finished, the SeD returns the result to the client." [BOLZE and al., 06]

Summary

In this chapter we have seen the reason why a single agent system can make problems and how this problem has been solved in DIET. We have seen the main architecture of DIET, the parts of the master agent, local agent, the client and the SeD and the interaction of these components. We have seen the initializing procedure to start a DIET instance and what's happen when a request is sent.

Chapter 3

The scheduler

Grid applications can be sequential, parallel or distributed and each one of them can generate resource-intensive requests. These applications are simultaneously executed on the grid and have to share resources. Meanwhile, these applications are trying to force the performance potential of the grid to speed up their own execution. That is why scheduling is crucial for high performance. The performance of a system cannot automatically be associated with high system throughput or any other throughput, because performance is a generic word and requires a context for an adequate definition. We will discuss the meaning of performance in Chapter 4.

We will briefly introduce the mechanism of scheduling, the different approaches to scheduling requests in a grid and we will see how scheduling is used in DIET.

3.1 What scheduling means

The adequate scheduling of requests is crucial for high performance grid architectures because it allows an optimization of the job execution and a load balancing for the different servers. For Francine Berman [BERMAN, 99] a high-performance scheduler consists of the following activities:

1. Select a set of resources capable of treating the application requests.
2. Assign application task(s) to compute resources.
3. Distribute data or co-locate data and computation.
4. Order tasks on computation resources.
5. Order communication between tasks.

In the literature, item 1 is often called *resource location*, *resource selection* or *resource discovery*. Item 2 is often called *mapping*, *partitioning* or *placement*. Additionally, grid computing allows distribution of problems over many servers, but sometimes some dependency between requests persists

by splitting a problem into multiple requests. That is why it can be necessary[1] to perform item 3. The scheduler must determine a performance-efficient distribution or decomposition of data

So starting from these five steps, a scheduling model of four components is proposed in the article [BERMAN, 99]:

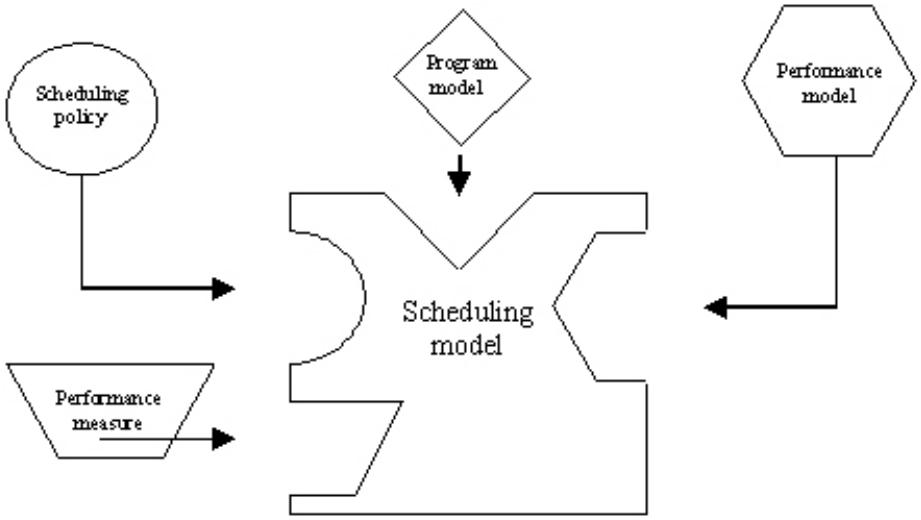

Figure 3.1: The components of the scheduling model

1. The **scheduling policy** is a set of rules for producing schedules. These rules describe which set of resources has to be chosen to achieve the performance goal. Simple examples of such policies are first come, first served, a preemptive policy, a fair queuing policy, a load balancing policy, etc. A concrete example of such a policy is DIET's scheduling and is explained in Section 3.3.

2. The **program model** abstracts the set of programs to be scheduled.

3. The **performance model** is used to evaluate the performance potential of the candidate schedule by abstracting the behavior of the program on the underlying system.

4. The **performance measure** describes the performance activity that must be optimized by the performance model.

"Schedulers employ predictive models to evaluate the performance of the application on the underlying system, and use this information to determine an assignment of tasks, communication,

[1] But this step can be needed for many more situations than for the case above. For example, for a serial execution, if the job is not going to be run where the data are located, the data will have to be moved

and data to resources, with the goal of leveraging the performance potential of the target platform." [BERMAN, 99]. Only a perfect interaction of these components enables high performance scheduling. An error in one of these elements will often slow down or even bring down the whole grid system.

The performance model, program model and performance measure are explained in Chapter 4 because they are mainly important for the performance evaluation.

3.2 Typologies

We can classify schedulers by using one of the following three criteria. The first criterion differentiates the schedulers by their performance goal (the scheduling shall optimize which component of the system?). The second criterion distinguishes centralized and distributed scheduling. The third criterion distinguishes schedulers by their approach to treating incoming requests. The design of a performance evaluation approach may be influenced by the type of scheduler. For example, the needed information for scheduling may change depending on what scheduler type is present. And this again may modify the performance evaluation.

Criterion 1: The different performance goals

A scheduler is used to optimize the performance of the grid system. Nevertheless the performance is always evaluated on a specific component of the grid. So each scheduler will optimize the scheduling for the grid element it is responsible for. This is the scheduler's performance goal. Different performance goals are pointed out here:

- The first performance goal is to promote the performance of the system. These schedulers are called *job schedulers* (**high-throughput schedulers**) and they optimize the job throughput of the system. As more jobs are executed by the system in a unit of time, system performance is improved.
- The next performance goal is achieved at resources: For example, *Resource schedulers* will try to ensure fair access to resources, that is, that all requests are satisfied. In another example, the resource scheduler will also try to optimize the resource utilization (the provided resources will be used to the maximum).
- Finally, the third performance goal is to achieve performance on individual applications: The *application schedulers* (**high-performance schedulers**) promote the performance of individual applications by optimizing performance measures like *minimal execution time*, *resolution*, *speedup*, or other application-centric cost measures.

The first and second scheduler types will promote the performance of the system above the performance of individual applications. These goals may conflict with the goal of the third type of scheduler. An example is using a high-throughput scheduler. It prefers smaller jobs and the bigger

jobs will never be executed. The job throughput is quite good, but the performance of the application which sends the bigger jobs is very low.

Criterion 2: Centralized and distributed scheduling

Two major types of scheduling are used in grids when talking about the set of information that the schedulers can access. The first one is centralized scheduling and the second one is distributed scheduling.

- Centralized job scheduling
  There is one central point in the architecture where information is collected and where the scheduling is performed. The scheduler has an overview of the whole virtual organization.

- Distributed job scheduling
  It can be advantageous to distribute the work of determining the appropriate schedule for a workload across the computational platform. Every agent includes a scheduler and decides for his subset of servers which one should be used. The work load is not concentrated on one single agent. This approach allows an application with high efficacy and scalability.

Criterion 3: Online job and Window-based scheduling

The incoming requests can either be scheduled by an online job scheduler or by a window-based scheduler.

In an **online job scheduling** approach each task is scheduled directly and there are no request queues at the MAs or LAs. The requests are scheduled as quickly as possible upon their arrival, which implies a very low scheduling latency. But on the other hand, it is possible to have a high load on servers. The cause is a long request queue on the SeD resulting in poor load-balance.

The on-line scheduling limits the scheduler's ability to adapt the schedule decisions and, under high-load conditions, this leads to scheduling too far in the future. In two cases this becomes a problem:

- There is no control over the number of jobs fulfilled. If a server suddenly becomes highly loaded, the requests already delivered to that server can not be rescheduled.

- Once the tasks are scheduled, the stored data needed for the queued tasks could be deleted by another process from the server's memory and disk. In this case it would be necessary to ask another server and this implies more work load on both servers.

If multiple tasks are scheduled in a short time interval, they can be scheduled on a server before the first one starts, thus introducing a lag in available data. The nonexistence of a history of schedules on agent level contributes to this low reactivity as well.

A third disadvantage is the impossibility of reordering the requests (i.e. by priority, affinity or by data requirements). For example there is no way to prioritize paying users.

Windows-based scheduling allows the agents to store multiple requests. In this way we achieve request flow control and task assignment re-ordering. For example, the re-ordering can be used to allow fairness to users (one of the concerns of DIET because it is a multi-user system), to accommodate inter-task and data dependencies, and to avoid the co-scheduling of multiple tasks on the same resource when the requests arrive nearly simultaneously at the scheduler. Figure 3.2 shows such a windows-based scheduling approach at the master agent and SeD level.

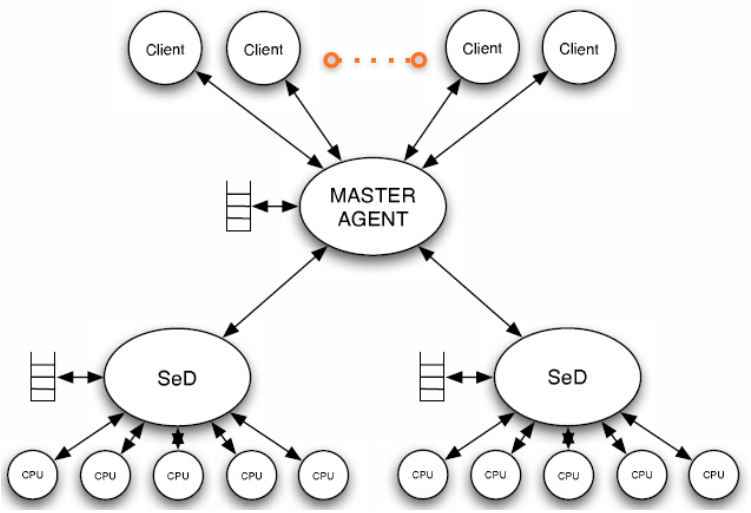

Figure 3.2: DIET extensions for request flow control by window-based scheduling [CARON and al, 05].

3.3 Scheduling in DIET

Most of the following information is extracted from the DIET User's Manual and the articles [DAIL and al., 06] and [SU, 05].

This section will explain the DIET scheduling approach. At this time, DIET's default scheduler is an application scheduler with focus on minimal execution time for requests. It is a distributed scheduling and DIET supports online job scheduling as well as windows-based scheduling (the second scheduling type is only available on explicit demand).

So this section will explain the default scheduling approach in DIET. This section is widely inspired by the DIET User's Manual [BOLZE and al., 06] and for the extension of DIET by the article [DAIL and al., 06]. When a task is submitted to the DIET system for processing, the following routine will start:

1. As we have seen in the general routine in paragraph 3.1 item 1 consists in the selection of SeDs that offer the service. And as we have seen in the Chapter 2, when DIET SeDs start-up and register with their parent, they specify which services they are able to treat. This information

is promoted in the hierarchy up to the MA. In this way an agent is aware of the offered services of each sub-tree.

When a request arrives at the client's master agent, the request will be forwarded to the sub-trees offering the service. In this way the request will be forwarded down to the SeD. With this approach it is not necessary to know if a child can treat the problem, it suffices to know the service is available *via* the child.

2. Each connected SeD can provide some information about itself. This set of status information is explained in detail in Section 4. This information can have a significant size that can cause unwanted network load. To avoid this problem, the bulkier detailed data generally remains at the server level in DIET and only synthesized information is passed up to the server's parent. The information is gathered by the SeD and passed up the tree only upon request (unlike NetSolve [ARNOLD and al, 01]), an approach that implies less overhead for the system in some conditions.

3. Upon receiving server responses from its children, the agent performs a local scheduling operation called *server response aggregation*. Effectively, candidate SeDs are identified through a distributed scheduling algorithm based on pair-wise comparisons between the status information of each SeD. The agent sorts these responses in a manner that optimizes certain performance criteria. The end result of the agent's aggregation phase is a list of server responses, sorted according to the aggregation method in effect. Until the responses reach the head of the hierarchy, the list is passed to the agent's parent agent to be aggregated with other lists.

4. The agent transmits the address of the SeD to the client.

Additionally, some extensions are implemented in the DIET scheduling policy. They are not used by default and we note them here for the sake of completeness.

1. In default scheduling the SeD will start to compute immediately when it receives the request. This resource sharing can be very harmful for resource-intensive applications. That is why the number of concurrent jobs at the SeD-level can be limited. In a simple first approach we allow SeD-level queues. They are now implemented in DIET and are illustrated in Figure 3.2.

2. As we have already mentioned in Section 3.2, it is now possible to use window-based scheduling. The MA stalls requests and stores them in a queue. At the appropriate time, the MA will schedule them as a batch.

3. These extensions have parameters that have an important impact on scheduling. That is why the most of these parameters [2] are configurable by the DIET service developer.

[2] Parameters like the allowed number of jobs that can run simultaneously on the SeD, the number of requests scheduled in a window and the time interval spent between windows

3.4 Scheduler's needs

Which performance data is needed by the scheduler? As every scheduler has its own objectives (low response time, high job throughput, ...) an unique answer will not suffice: context-sensitive performance evaluations must be provided.

As DIET is a system intended to support many kinds of applications and as the user has the opportunity to create his own scheduler via the use of plug-ins (explained in Section 3.6), it is important to take into account these complex needs.
We have seen that the data used by the scheduler depends on the performance goal. So the Holy Grail would be to know the corresponding performance already for each server before the execution has started. This capability would make it possible to develop a schedule that always chooses the most accurate server. We call this performance foresight for a specific application an **application-specific performance prediction**. In general, performance predictions are based on the assumption that observation of past performance implies that future performance levels will be similar. Nevertheless these predictions are not always possible to produce, so some other performance evaluation techniques must be available. Performance evaluation is described in detail in Chapter 4.

3.5 Using resource information services in DIET scheduling

It is now important to understand the difference between the scheduling operation itself and the information used by the scheduler. Scheduling depends on performance information provided by the system. Not only SeD status information could facilitate the decision making during scheduling, but also inter-architecture data like network bandwidth between SeDs. The *Network Weather Service* (NWS) [WOLSKI and al, 99] and the *Fast Agent's System Timer* (FAST) [QUINSON, 02] can provide this information. DIET implements functions to access these two modules.

The DIET scheduling policy uses the different types of SeD-provided data. Each data type results in one comparison level with an appropriate aggregation function. The different levels have a special ordered sequence. At first, level one will be used. If the data type is not available for either SeD, or the comparison shows that the SeDs are of equal performance, the next lower level will be compared.
This approach prioritizes SeDs that have this first data type. In this way it ensures that a scheduling decision can always be made. It is not possible to compare two different data types. The following sequence will explain the different levels and represents the behavior of DIET:.

1. **FAST**: SeDs compiled and properly configured with the performance evaluation module FAST [QUINSON, 02] are able to provide *application-specific performance prediction*. FAST's data consist in a prediction of how long the computation of the special task will take on the specific SeD. The aggregation function is a simple minimize function of this value.

2. **NWS**: SeDs compiled and properly configured with NWS are capable of providing *resource performance forecasting*. The forecasting consists amongst others[3] of a value indicating the fraction of CPU available for new processes in percent (*freeCPU*) and a value indicating the amount of free memory in megabytes (*freeMemory*). These two values are first weighted ($freeCPU^3$ and $freeMemory^{0.5}$) and then combined in the following formula to give more basic performance estimation:

$$Performance_{byNWSdata} = \frac{1}{freeCPU^3 * freeMemory^{0.5}} \tag{3.5.1}$$

   As we can see the weight given to freeCPU is higher than the weight given to freeMemory. The aggregation function is the maximization of this estimation.

3. **Round-robin**: If neither FAST nor NWS provide performance data, a round-robin approach will be used to achieve probabilistic load balance. When a server is assigned jobs for execution it records the time for the last job as a time stamp. On demand the SeD can now compute the time elapsed since last execution. The aggregation function here is the longest elapsed time.

4. **Random**: Sometimes the SeD is unable to store the last time, the DIET scheduler will chose randomly between two otherwise equivalent SeD performance estimations.

3.6 Plug-in scheduler

The most difficult challenge for scheduling is to determine a good policy. In the example of the default scheduling approach with NWS data, we already see two problems showing that the aggregation function is not always applicable. By using the formula described in Section 3.5, the scheduler will always prefer SeDs with high freeCPU and high freeMemory. Other parameters that may influence the execution time like the network, disk speed or CPU frequency are< not taken into account. Additionally, the weights attributed to freeCPU and freeMemory do not reflect their importance in every application. Some applications are more memory demanding and some others much more CPU demanding. As [BERMAN, 99] indicates, "it may not be impossible to obtain optimal performance for multiple applications simultaneously".

The **plug-in scheduling** facilities are designed to allow DIET service developers to define application-specific performance measures and to implement corresponding scheduling strategies. In this way the plugin permits maintenance of the distributed scheduling design. It enables an extensible performance measurement system and tunable comparison/aggregation routines for scheduling.

One of the performance measurement systems is the new CoRI management system for resource information (see Chapter 5). Other systems like FAST or NWS can be used too.
The plug-in in conjunction with these measurement systems enables various selection methods for

[3] Additionally it is possible to receive the network capacity between two entities. But this information is not yet used in the default scheduling of DIET.

using basic resource availability, processor speed, memory, database contention, or any other measurable information as criteria for the scheduling. The aggregation routines can be modified or rewritten to allow an optimal scheduling.

The plug-in includes a scheduler API, different metrics (last execution time, CPU load and free memory capacity via FAST, ...), and different aggregation functions (the maximum and minimum of a criterion value,...).

Summary

In this chapter we have seen a brief introduction to scheduling in DIET, some of its importance and some of its problems. Also we have seen the online distributed application scheduling implemented in DIET, its actual order of comparison levels and its extension, the plug-in scheduler.

Chapter 4

Performance evaluation

The software and hardware resources of the underlying system may exhibit heterogeneous performance characteristics, resources may be shared by other users, and networks, computers and data may exist in distinct administrative domains. Appropriated scheduling is only possible if exact information about the state of all machines is provided. That is why a good performance prediction tool is crucial to take the full advantage of a grid system.
We will compare the scheduler's need for status information with the set of information actually provided by the tools used in DIET. Finally, we will cite the reasons for the new tool CoRI in DIET .

4.1 What performance means

Until now in this dissertation, the term "performance" was not precisely defined. We know that performance is a non-functional requirement of the majority of systems in general and especially in grid systems [BERMAN, 99]. The term is used in many domains, and we will define it only in the grid computing context.

Nevertheless we will take inspiration from the human management domain [SONNENTAG and al, 02], where performance is a critical requirement as well. The parallels between both domains seem very close. For example, managers have to schedule the jobs to their employees in the same way as the MA must schedule the tasks to the SeDs. And in both systems this scheduling must achieve high performance. In the human resource field, only actions which can be scaled, i.e., measured, are considered to constitute performance ([CAMPBELL and al, 04]). In grid computing, we can find the same approach ([JAIN, 91] and [BERMAN, 99]). And in order to allow measurements, we have to associate the performance with criteria. The latter are evaluated in **metrics** that will measure performance in a concrete manner. Here we will list commonly used performance metrics [JAIN, 91] to give an idea what performance can mean. We mention here that the definitions of these metrics are only one of many possibilities.

- The **response time** is defined as interval between a user's request and the system response. This definition is simplistic because request and response are not instantaneous. Typing the

request and outputting the answer take time. Different solutions are possible by defining the beginning and the end of the interval to measure, but we will not detail this definition here.

either the interval or as the interval between

- The **reaction time** is measured as the time between submission of a request and the beginning of its execution by the system.
- The rate (requests per unit of time) at which the requests can be serviced by the system is called **throughput**. For example, the Millions of Instructions Per Second (MIPS), or Millions of Floating-Point Operations Per Second (MFLOPS) measure the CPU throughput.
- The **utilization** of a resource is defined as the fraction of time the resource is busy servicing requests. It can be calculated by the ratio of busy time to total elapsed time over a given period.
- The probability of errors or the mean time between errors defines the **reliability** of a system.
- The **availability** of a system is defined as the fraction of the time the system is available to service users' requests.

4.2 Which elements may affect performance?

If we want to measure the performance it would be necessary to specify which elements in the system may have an influence on the performance. Or, we can ask in the other sense as well, which are the system components that we want to evaluate? Hence we will list some of these system elements. Note that this listing is not necessary exhaustive.

1. The first element is the workload. **Workloads** are the requests submitted by the users to the system (for example the network workload would consist amongst others of the transmission of files over the network).

   The performance can be influenced by system and workload characteristics. These characteristics are called **parameters**.
   For example, **system parameters** may include CPU operation number (for CPU allocation), request size (for network allocation), or working set size[1] (for memory allocation).

   Workload parameters may include the number of users (this indicates the number of active application entities that are concurrently engaged in the system), priority, request arrival rate and distribution (this indicates the number of requests generated per unit time)

2. Since grids are built in a heterogeneous environment, the performances of the different machines will be influenced by hardware and software elements.

   We will list here some important **hardware** characteristics, but this list is not necessary exhaustive. Because every application and every machine is different from each other, it is

[1] The working set size is the set of virtual memory pages currently used by the process.

very complex to take into account each element. Remember that each of these elements *might* be a metric for performance evaluation, but this will be discussed in Section 4.3.4. We will group them into categories for a better overview.

Central Processing Unit (CPU) Different CPU characteristics can influence the performance of the CPU:

- The CPU frequency,
- the instruction set supported by the CPU,
- the number of CPUs,
- the cache level 1 (called cache L1, small and fast memory near the CPU),
- the cache level 2 (called cache L2, for older processors: the external memory situated on a separated chip near the CPU; for newer processors: a second internal memory like L1),
- the cache level 3 (called cache L3, only available for newer CPU generations: previously called cache L2, the external memory situated on a separated chip near the CPU),
- the frequency of the cache,
- the frequency and theoretical bandwidth of the front side bus (FSB, bus between CPU and RAM, BIOS, hard disk,...), and of the back side bus (bus between CPU and cache L2)

Memory (RAM)

- The RAM frequency,
- the RAM type (DRAM (dynamic RAM), SRAM (static RAM), DDRAM (Double Data Random Access Memory), FPM (Fast Page Mode), ECC (Error Correcting Code) and its bandwidth (maximal transfer from RAM to the L2-cache or to the L3-Cache), EDO (Extended Data Output), SDRAM (Synchronous Dynamic RAM)),
- the RAM capacity.

Virtual memory (SWAP)

- The SWAP size
- The SWAP's storage type (hard disk, flash, ...)

Disk

- The maximal and actual disk rotation,
- the number of devices,
- the number of read-write heads of each arm

- the cache size,
- the technology of read and write (GCR (Group Coded Recording), MFM (Modified Frequency Modulation) , RLL (Run Length Limited), PRML (Partial Response/Maximum Likelyhood), EPRML (Extended Partial Response/Maximum Likelyhood),. . .)
- the disk size,
- the number of platters,
- bus types (for example ATA (IDE, EIDE), Serial ATA, SCSI, SAS, FireWire (aka IEEE 1394), USB, Fiber Channel,. . .),
- the RAID version,
- characteristics of other peripheral devices (CD-ROM, DVD, streamer, NFS,. . .).

Network

- The theoretical bandwidth (for example for Ethernet 10, 100, 1000, or 10000 Mbit/s)
- the number of hops between the two edges points.
- the protocol used
- many more,. . .

Additionally to this hardware aspect, the **software** can influence the performance too. The simple difference of the version can have several important impacts on the metrics like the response time, the reliability of the system, even the power consumption or other criteria. In addition to versions, one program could be faster than another one. Of course, the operating system can have an effect on the performance too. There are a lot of other reasons why software can affect performance, but our main concern in this dissertation will be the hardware aspect.

3. It is important to take into account the different system components, for example the servers, the agents, the network, the databases,. . .

4. Even the **previous scheduling** can have an effect on the performance. Workloads can be placed on the SeD already and so the second arriving workload has to share the resource capacities with the other scheduled workloads.

5. If application **data** is present at the SeD, it is not necessary to request them via the network. Further, the simple fact that the data is in the memory could increase performance because the access is faster and easier than if the data would be on disk.

6. **other elements** can influence the performance. As the cited elements are primarily concern of performance in the meaning of reaction time and response time, we have to remember that it is extremely difficult to cite every element of every kind of performance.

Furthermore, we can distinguish between **static** and **dynamic** elements that could affect the performance. Static elements need to be evaluated only once, and dynamic information must be evaluated constantly. Hence prediction performance must be adjusted constantly too. Nevertheless we have to take care of declaring elements as constant because new technologies could make them dynamic. For example the number of CPUs seems to be static, but in reality some systems support the dynamic adding or removing of CPUs. Another example is the the disk rotation: it is possible to decrease the speed for reducing noise and power consumption.

So, detecting the machine with the best performance is very difficult, because a lot of parameters affect the performance. Later in this chapter we will discuss the utility of all these characteristics. Therefore, different approaches for performance evaluation are elaborated and they are explained in the next section.

4.3 The Grid Resource Information Service

In the context of grid systems, information services are any kind of services that returns information about the status of the system. For example, one application-specific performance prediction service predicts the computation time for a request on a SeD. Another information service is the resource performance forecasting service that gives some performance forecasts about resources. In this chapter we will examine the different levels of information services in a top-down approach. Firstly, we will briefly introduce the different levels and their interconnections, and afterwards we will discuss each level in detail.

We have seen in Chapter 3 that performance prediction is the best information for the system for optimizing the scheduling. But in the majority of cases this prediction needs, amongst others detailed knowledge of the service behavior. Additionally, the prediction would need detailed information about SeD's resources. This information depends on two factors:

1. The status of the architecture: The workload by other requests must be in included in the prediction. If the scheduler knows that five other requests are executed on the SeD, it must take into consideration this fact for its prediction.

2. But the first point has limits because we are on a grid and nevertheless one of the main characteristics is that it is a concurrent environment. This drives us to a major problem namely the impossibility to know how much the SeD is really occupied. To counteract this problem we introduce here another important component of a grid, namely the **resource performance forecasting**.

So the prediction tool gets resource performance forecasting that uses numerical models to generate forecasts. We will introduce their principle and some tools in Section 4.3.2.

But the resource performance forecasting needs *dynamic system availability data* about the SeD, the network and any other components that are present in the architecture. Therefore they use **resource monitors** that record different measurements data of different resources. These resource monitors are explained in Section 4.3.3.

These measurements are achieved by **measurement services** that are placed and executed nearby the components. They are explained in Section 4.3.4.

So, each one of the cited services above represents one **data level** of the performance evaluation. Measurement service provides elementary information, monitors combined and historical information, predictions services provide complex estimations of resource or application.

Figure 4.1 shows an example of architecture that shows the different elements cited above. In this example, the application-specific performance predictions at the SeD depend on three different resources. Therefore each resource is linked to one sensor and one monitor. As we can see for the resource monitor C, it is possible that multiple resource performance forecasting tools simultaneously access the resource monitor.

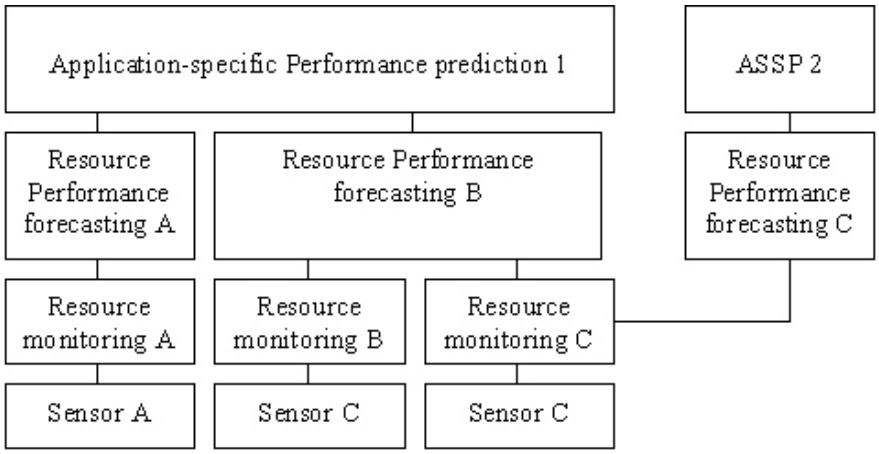

Figure 4.1: **Performance prediction** This figure shows an example of a performance prediction model with its four levels, namely the application- specific performance prediction, the resource performance forecasting, the resource monitor, and the sensor.

Further it is possible that the scheduler avoids the performance prediction tool and that it asks the resource monitor directly. So the presented architecture is only one model that contains the different components of performance evaluation, and not all developed tools make the clear distinction between these levels, or implement all levels. It is even possible to use a monitor above performance predictions, in this way this monitor will collect predictions coming from different performance prediction tools.

Nevertheless it is not trivial to get performance prediction services, because many levels must be used for providing an exact prediction to the scheduler. We can remark that the interconnection and dependency to the underlying levels is very high. We will now analyze the different elements.

4.3.1 Performance prediction

"*Knowledge that does not include the future is not knowledge at all.*"

Hans-Peter Dürr (*1929), German physicist,
1987 Alternative Nobel Prize.

Performance prediction is widely used in grids and many approaches are elaborated in this domain. We will not entering into the details here, because the approaches can be very different and this is not the main topic of this dissertation. We will only introduce the approaches that are important for our next step. As we have already seen in Chapter 3, the scheduling consists of four elements, namely the scheduling policy, the program model, the performance model and the performance prediction model. The first element, i.e. the scheduling policy, has been explained in Chapter 3. The three other elements are used for the performance prediction of application,

Program models

When problems are too big to be computed on one machine, they will be divided into small tasks. As a task can still have dependencies on other tasks (for example one task needs the result of another), the parallelism is not perfect. So these communicating tasks have to be modelled for assuring high-performance and scheduling. Several approaches are elaborated and some of them are cited in the article [BERMAN, 99]. Nevertheless, we mentioned this component for the sake of completeness, but shall go no further, as this is another branch of research.

Performance model

The goal of performance modelling is to get a better understanding of a computer system's performance on various applications. One use of this model can be to create a performance prediction for each machine and then to use this knowledge to evaluate the performance potential of a given schedule.

To generate such a performance model we can use different measurement and analysis techniques that we will describe later in this section. In general, it provides an abstraction of the behaviour of the application conditioned by the underlying system, and this abstraction will be described in a compact formula.

Once this model is described, it can be used to generate a performance prediction for an application. Therefore, we only have to specify resources, capabilities (e.g., maximum floating point rate or available bandwidth), and problem parameters.

Firstly, we will introduce some of the many approaches that are elaborated for creating performance models. Therefore each approach is described by three aspects:

- who supplies the performance model (the system, the programmer or a combination of both),
- its form,
- and its parameterization (static and dynamic information)

We shall will describe some of the approaches in very general terms, because these techniques are very mathematical and our goal is not to discuss performance models, but only to introduce them so as to understand their needs better.

So, on one end of the spectrum, there are **scheduler-derived performance models**, which require little intervention from the model's user. Some applications use coordination language and derives in this way "skeleton" performance models from programs. Others use the last program iteration for building a program dependency graphs or simple benchmarks. As a first example we can mention FAST that will be explained in Section 4.3.1.
Other application-specific performance prediction tools are SPP(X) [AU and al, 96],
MARS [GEHRINF and al., 96], Dome [ARABE and al, 95], ...

At the other end of the spectrum are **user-derived performance models**. Often the program assumes that the performance model – and sometimes even the resulting schedule – will be determined by the user. Then, the models are parameterized by static and dynamic information into a prediction of application performance. For a concrete example, we refer to the AppLeS project [BERMAN and al., 97], and the I-SOFT scheduler [FOSTER and al, 96].

It is worth mentioning that some approaches **combine both programmer-provided and scheduler-provided performance components**. These approaches require both programmer and scheduler information.

To summarise, the different models of the different approaches are built in a certain manner, but then they will be parameterized by using static and dynamic information. For more information about prediction model we refer to the article [BAILEY and al, 05].

Performance prediction characteristics

What should the qualities of a performance prediction be? In general (according to [BERMAN, 99]), performance prediction models should have three main characteristics:

- **Time-specific** The performance delivered by the system resources vary over time in the grid, hence the performance predictions must also vary over time and should be timeframe-specific. This implies that the predictions have a limited life.

- This calibration is done if the prediction utilizes **dynamic information** to represent variations in performance. Since concurrent access to resources is common in computational grids, application performance could vary enormously over time and per resource. Dynamic parameters are utilized to reflect the evolving system state of the grid, to help models to perceive Dynamic information about performance variations.

- Since the heterogeneity of its components is one of the major characteristic of the grid, the scheduling must be able to adapt its behaviour to a **wide spectrum of potential computational environments**. Hence the performance prediction models must be able to include distinct execution environments in their evaluation.

The performance prediction tool FAST

This section deals with FAST [QUINSON, 02], a performance prediction module that can be used in a grid environment, and particularly in DIET. Firstly we will show what kind of information FAST can provide to the user, then we will explain the principles of its prediction, show its architecture, and finally its dependencies on other software. This section is based on [CARON and al, 05].

FAST can provide SeDs with improved performance prediction capability. In fact, FAST provides the following information to the user (the "user" is the scheduler in our context):

- a forecast for the data transfer time between two FAST-enabled machines,
- a forecast for the time and space needed to solve a problem with given problem parameters and given set of computational resources.
- the combination of these two quantities,
- other lower level system availabilities (the free CPU, the free memory, the number of CPUs, the latency and bandwidth of any TCP link). This information is not used as performance prediction, but constitutes a lower data level included in the FAST tool.

The prediction is based on two different data sources:

- Static data acquisition is used to predict the time and space requirements of the routine. Several approaches are used, depending on the type of routine.

  1. Routines that can be measured easily (numerical algebra routines whose performance is not data-dependent and where a clear relationship exists between problem size and performance) are match-marked in time and space complexity. The resulting data is fited by polynomial regression and is stored in a database. These operations can be time consuming but they are executed only once when the routines are registered at the agent.
  2. All the routines cannot be evaluated with this approach. Indeed, some can be too difficult to evaluate experimentally. They use concurrent execution of functions[2] and for this reason; too many parameters have to be taken into account. In this case, FAST allows the developer to specify the computation time by a complex expression that aggregates the computation time of other routines (obtained with the first method) to the communication time of the parallel version of the algorithm given by a careful evaluation of the algorithms.
  3. A final approach makes some problems for FAST: the kind of routines whose performance depends on characteristics which are hard to extract (like the shape of the matrices involved or the values of their elements). The processing of such routines is not yet solved in FAST and will need further work.

[2] For example the SCALAPACK library and the function `pdgemm`

- The dynamic data acquisition is the second module and can acquire system performance capabilities at runtime. It allows FAST to take into account the actual work load of the resources.

The use of these two data types enables FAST to provide accurate forecasting to the client application. Figure 4.2 gives an overview of the FAST architecture, which is composed of two main parts. On the bottom of the figure, the **benchmarking** program is used for the routines of the first type. Then, on top, the figure shows the FAST **library** that is divided in the two sub-modules already explained before. FAST uses two types of external tools principally (in grey): the system

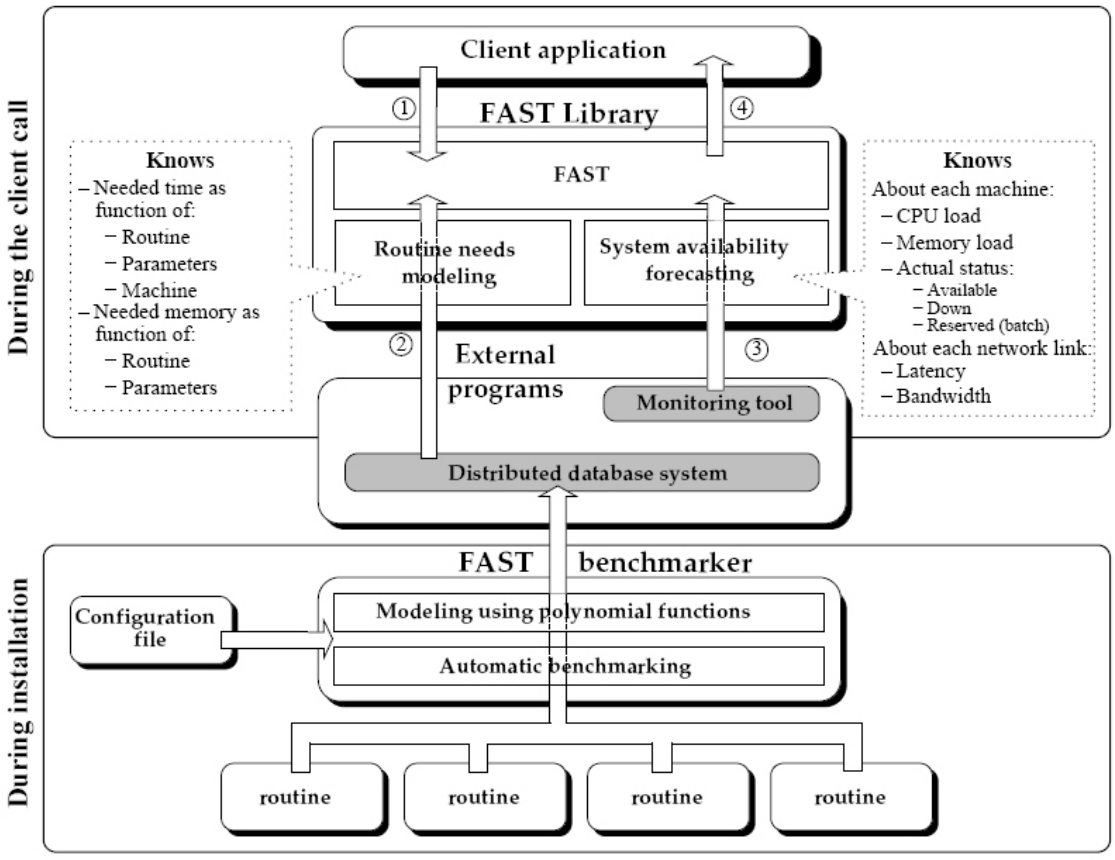

Figure 4.2: **FAST's architecture [CARON and al, 05]**

monitoring tool (NWS) is used for the dynamic data acquisition, and the distributed database that stores the sets of static data computed at the installation phase about routine's needs.

So the dependent programs are:

- **NWS** the Network Weather Service
- **GSL** the GNU Scientific Library

- **OpenLDAP** an implementation of the Lightweight Directory Access Protocol that allows the storage of the static data.

Although the primary targeted application class consists of sequential tasks, this approach has been successfully extended to address parallel routines as well, as explained in more details in [CARON and al, 02]. One of the advantages of FAST is that since performances prediction is performed only in the DIET SeD, no modification is needed to the client code. A second advantage is that monitoring new resources like free disk space or non-TCP links should be relatively easy in the FAST framework because the API is quite easy to use.

For more details about FAST, the reader can visit the FAST Reference Manual[3].

4.3.2 Resource performance forecasting service

Resource performance forecasting services provide performance prediction of resources. In contrast to application-specific performance prediction, they are independent on the applications that run on the system. For this reason these services are often[4] used in performance prediction services for calibrating their predictions. So, there is a high dependency of the application-specific performance prediction services on resource performance forecasting services.

NWS

In this section we will introduce the forecasting method of NWS. The Network Weather Service (NWS) [WOLSKI and al, 99] is a distributed system that periodically monitors and dynamically forecasts performance of various networks and computational resources. It is used by many grid projects like AppLeS [BERMAN and al., 97], Globus [GLOBUS], NetSolve [ARNOLD and al, 01], Ninf [NAKADA and al, 99] and DIET.

Firstly, we will introduce the architecture of the tool, and then explain the principle of resource forecasts in NWS.

NWS is built upon the two lower information service levels, namely a monitor service collecting all information from sensors, and the sensors extracting the raw performance data from the component. That is why we will repeat NWS in the sections 4.3.3 and 4.3.4.

By using these information services that capture the current state of each platform, NWS forecasts the evolution of the monitored system in short-term and characterizes the performance deliverable at the application level dynamically.

The current NWS forecaster needs two elements for generating forecasts of future measurement values.

1. The first element consists of pairs of time stamps and measurements that are ordered by time, called time series,

[3] `http://graal.ens-lyon.fr/FAST/docs`

[4] indeed, some approaches (for example [GAUTAMA and al., 00] do not take into account the environment, that is why they are inadequate for the grid computing

2. The second element is a set of forecasting models that can "predict" the measurement based on the measurements that come before it in the series. They consist of statistic functions like average or median.

So, this set is now applied to the pairs, and then the forecasting technique chooses the most accurate model. Nevertheless, this technique does not incorporate any modelling information specific to a particular series. As each model will produce a prediction, the most accurate model is the one that has the lowest cumulative error on its prediction.

The advantage of this adaptive approach is that it is ultimately non-parametric and, as such, can be applied to any time series presented to the forecaster. While the individual forecasting methods themselves may require specific parameters, different fixed settings are included for particular methods with the assurance that the most accurate parameterization will be chosen.

Other forecasting methods

Other researches were made about resource forecasting. Some methods are listed here:

- Semi-Nonparametric Time Series Analysis (SNP) [GALLANT and al., 92],
- automated Box-Jenkins[5],
- wavelet-based models [OGDEN, 97].
- The resource prediction system(RPS) [DINDA and al., 99].

4.3.3 Monitoring services

"To measure the performance of a computer system, you need at least two tools - a tool to load the system (load generator) and a tool to measure the results (monitor)" [JAIN, 91]. Loading the system can be performed by requests (in this case it would be an online loading) or it can be done by special programs in order to simulate the load of a machine (called benchmarks). The first element will be explained in Section 4.3.4. The second, namely the monitoring, will be explained in this section.

In the literature, monitors are not only able to **manage the resource information sources** (often called producer [TIERNEY and al., 00], [ANDREOZZI and al, 03]), to **store** the resource information coming from the producer and to **manage the access** to this information by the consumer (in our case the upper levels); they are also responsible for producing performance predictions of applications and resources (seen in sections 4.3.1 and 4.3.2) and for measuring the system (this will be seen in Section 4.3.4. But we have implicitly made a distinction of these different functionalities for a better understanding.

In fact, the measurements are not generated by the monitor, but are collected from a lower level measuring service, individual sites, files, programs, web services or other network-enabled services.

[5] `http://www.autobox.com/index.html`

Furthermore, the monitor can also use other monitors, and a hierarchy of monitors is used to gather information. This monitoring data can be used to achieve three goals:

1. It is used to determine the source of performance problems (bottlenecks,...).
2. The fault detection and recovery mechanisms need the data to determine whether a service is down.
3. And finally, it is used for resource capability-aware scheduling, which will tune the system and/or application performances.

We can see that these systems have different application ranges, so we will now cite the most important general requirements [TIERNEY and al., 02] for these kinds of applications.

- **Low latency**. As performance data are time-sensitive information valid for a short time interval only, their transmission must be done with a low latency.
- **High data rate**. As the sensors can generate performance data at high rate, the system should be able to support these operating conditions. Additionally, the sensors could send the data in bursts, so the system should be able to limit these rates in order to avoid overwhelming the consumer.
- **Minimal measurement overhead**. Multiple access to information must not be intrusive or the intrusiveness should be limited to a minimum.
- **Secure**. The access to the information should be subject to restrictions. The monitoring system has to control the access policies imposed by the owner of the data and it must ensure its own integrity.
- **Scalable**. Because we are working in a grid environment, the number of resources, services and application to monitoring, as well the number of consumers can be very high. That is why the monitoring system has to provide scalable measurement.

Several approaches are made for ensuring these characteristics and we will now analyse the architecture of NWS as an example.

The NWS monitoring system

The NWS Monitor allows the user access historical performance data of different measured system components.

Since NWS must be suitable for distributed applications, it seems logical to distribute the monitoring system as well. On account of this obligation, different NWS component processes are necessary for monitoring:

1. The **persistent state** process, called NWS Memory, stores and retrieves measurements from persistent storage. This mechanism allows a higher robustness because information stored at the sensor side could be lost on a failure of process memory at the sensor side. Hence a NWS

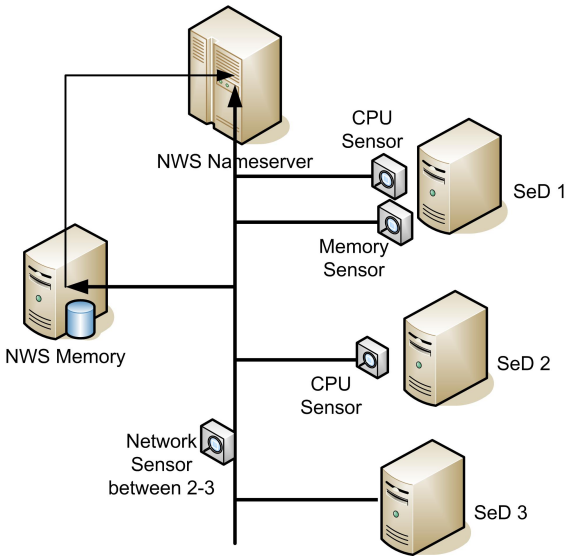

Figure 4.3: A simple example of the NWS architecture with several sensors, one NWS memory and the NWS name server

memory can be deployed for storing the measurements of the last time-intervals. Indeed, the measurements are not stored indefinitely, because they loose their utility progressively. It is possible to deploy a hierarchy with multiple memories which allows a distributed work load.

2. The **name server** process, called NWS Name server, implements a directory capability used to bind process and data names with low-level contact information (e.g. TCP/IP port number, address pairs). This name server is unique in a NWS hierarchy and every sensor and every memory of a specific hierarchy has to log onto the same nameserver.

3. The **sensor process**, called NWS Sensor, gathers performance measurements from a specified resource. It will be explained in the Section 4.3.4 after the introduction to sensors.

4. Finally the **forecaster process** was already explained in the Section 4.3.2. It produces a performance prediction during a specified time frame for a specified resource by using the information stored in the NWS memory.

Figures 4.3 represent the interaction between the different components. In this deployment sample, multiple sensors for CPU load, network load and memory capacity are subscribed and only one memory server is used.

Other monitoring services

There are a lot of other monitors, for example:

- **Ganglia** [MASSIE and al., 04] is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids.
- **Nagios**[6] is a host, service and network monitoring program.
- **Survivor**[7] is a monitoring service that uses a scheduler approach for monitoring. This allows an easy administration of the different simple characteristic requests.
- **REMOS**: The REsource MOnitoring System [DEWITT and al., 98] provides network information.
- **INCA**[8] is a flexible framework for the automated testing, benchmarking and monitoring of Grid systems[9] that includes automated testing, benchmarking, verification, and monitoring of service-level agreements at specified intervals. It is part of the globus alliance and it is used in TeraGrid[10]. Its architecture consists of a server that is responsible for managing data collection and making data available to consumers and clients responsible for gathering data from a resource.
- **Argus**[11] is a system and network monitoring application.
- **BixData**[12] is a cluster management tool that includes monitoring and system administration features. It monitors services (HTTP, ping, POP3, SMTP), performance, and processes.
- **RMI of Legion** [CHAPIN and al, 98] the resource management infrastructure deployed in Legion has similar goals as the grid resource information services in DIET.
- **MonALISA** A distributed Monitoring Service Architecture [NEWMAN and al, 03].
- **MDS** [CSAJKOWSKI and al, 01] is a tool for monitoring and discovery systems and is used in Globus. Amongst other it utilizes Ganglia, Hawkeye[13], WS-GRAM[14] for collecting information about resources.
- **R-GMA**[15] has another approach for grid information services: it makes all the information appear like one relational database. A producer-consumer approach is used for collecting and distributing the information.
- **JAMM** [TIERNEY and al., 00]

[6] `http://www.nagios.org/`
[7] `http://www.columbia.edu/acis/dev/projects/survivor/`
[8] `http://inca.sdsc.edu/`
[9] http://inca.sdsc.edu/
[10] `http://www.teragrid.org`
[11] `http://argus.tcp4me.com/`
[12] `http://www.bixdata.com/`
[13] `http://www.cs.wisc.edu/condor/hawkeye`
[14] http://globus.org/toolkit/docs/3.2/gram/
[15] `http://www.r-gma.org/`

- **Hawkeye**[16] Hawkeye is a network monitor.

For more information, the white paper [GERNDT and al., 04] and the article [XUEHAI and al., 03] list some of these monitoring tools amongst others.

4.3.4 Measurement service

The lowest layer for grid information services is measurement services. They probe resources for simple or composite metrics. These metrics will be available for the upper levels in form of raw data. It is up to the higher levels to store, filter, interpret, analyze, or compare the data.

Nonetheless, building this service is indispensable for the scheduler because without it, the scheduler would be blind to all activities of the grid. Therefore, some requirements are crucial for providing a good measurement service.

- The intrusiveness of the measure has to be limited to an acceptable fraction of the available resources.
- It is important that each resource sensor shares a common definition of the metrics. For example, the consumer would be confused if one sensor transmits the write speed of a hard disk in bytes and another provides it in megabytes.
- The sensor should be extendable for new grid environments. For example, if until now the sensor is used only for UNIX-based operating systems, and it should now be used under Windows XP which does not provide the same basic functions, the sensor should be able to provide these new functions.

Firstly, we want to introduce some techniques that are developed for extracting information. The first such technique, namely the **sensor**, is used to manage the data transmission to the monitor and to manage the measurement conditions (i.e. the frequency measurements are taken). The sensor does not measure the resource; it is rather like the person who uses a thermometer to measure the temperature. It uses other techniques, namely the **benchmark** and the **simple request**, for perceiving the data which it will send back to the monitor. Hence, we can see the sensor as a middleman between the monitor and the measuring instrument. We will now explain in details the sensor technique and afterwards the benchmark and the simple request.

Sensors

A sensor provides dynamic information about specific devices of a specific component present in a system. A device can be any element of the component that can influence its performance (cf. Section 4.2).

A sensor manages the queries which are used to consult the device. These queries consist of calls of simple request and executions of benchmarks.

Therefore, sensors are deployed at the component's side, often as an autonomous processes. Furthermore, the capture of the device state and the data delivery can be caused by the monitor,

[16] `http://hawkeye.sourceforge.net/`

but also by device activity or by an elapsed time period. For example the NWS sensor uses different basic functions and fixes the rate of polling the device. Furthermore, the sensor can generate events if dynamic thresholds are reached. For example, if the used memory exceeds a given amount, an event will be created and sent to the monitor.

Some sensor types:

- *Host sensors* perform host monitoring tasks, and provide information like CPU load, available memory, or TCP retransmissions.
- *Network sensors* perform queries on a network device, for example a router or switch.
- *Process sensors* observe the process status and generate events when the status changes (for example, when it starts, dies normally, or dies abnormally).
- *Application sensors* are embedded in applications and can generate events when static thresholds are reached. For example in DIET, the SeD could generate an event if the number of incoming requests exceeds a fixed number. Another goal for application sensors can be to collect information about the application performance, like the execution time or the used resources that can be used for further performance analyses.

Example: NWS sensors

We will introduce here the NWS Sensors to gain a better understanding of the role of sensors. More detailed information can be found in the articles [WOLSKI and al, 99] and [CARON and al, 05].

So the role of a NWS sensor is to gather and store time stamp-performance measurement pairs for a specific resource. Each sensor process may provide different performance characteristics of the resource that it observes.

In NWS, the architecture includes a distributed set of sensors with two types of sensors: **network sensors** that provide *small-message round-trip time, large-message throughput, and TCP socket connect-disconnect time* measurements of any TCP/IP link and **host sensors** that deliver the *CPU load*, the *available memory*, the *time-slice* and the percentage of *CPU power* a new process would get at startup, and the *disk space* on any host.

NWS CPU sensors use a combination of Unix provided utilities (namely `uptime` and `vmstat`) and CPU tests that periodically actively measure the CPU utilization. In this way, the sensor does not only provide the CPU occupancy time and an estimation of the available CPU for a new process, but also takes into account other more dynamic aspects like the priority of the other processes. Notice here that the stressing part of the sensor, namely the CPU test, is executed less often than the Unix utilities because its execution frequency is determined dynamically.

The NWS network sensor relies only on active measures because data between machines is not consistently available in end-to-end networks. The test consists of a timed network operation, such as the transmission of a fixed amount of data, which are performed at regular intervals.

Benchmarks

The sensors need benchmarks that measure the performance of the system by evaluating the performance in an active way. "Active" here means that the system's resources are used for this estimation: the resources are not available for other applications (or are available only to a limited extent).

We have seen in Section 4.2 that it would be difficult to estimate the performance of a system by a simple calculation.
For example we could weight each characteristic of the CPU (i.e. 0.8 × frequency + 0.1 × cache L1 size + 0.1 × bandwidth front side bus) that represent the weighted speed of the processor. Since the weighting would require a lot of knowledge about the different components; since some terms can not be defined in a general manner (i.e. the frequency which does not have the same meaning everywhere); and since there are always some aspects not taken into account (i.e. is it important that the CPU supports hyperthreading technology?), this model is by no means adequate for measuring the performance in heterogeneous grid environments.

For these reasons, the performance of a system is often evaluated by **test workloads**, which are workloads used in performance studies. Either they are **real**, observed during normal system execution, or they are **simulated**. The problem of a real workload is its non repeatability because analysing its results is complicated and therefore not suitable for use as a test workload. Instead, the synthetic workload (also called benchmark) simulates the characteristics of the real workload but can be applied repeatedly in a controlled way.

We want to introduce now some types of benchmarks. Most of them were developed for comparing processors and timesharing systems, nevertheless their general principles can be applied to other computing components such as network, databases, and so forth [JAIN, 91]. We will explain each of these benchmarks and discuss the circumstances under which they may be appropriate.

- **Addition instruction:** Historically, the addition instruction was one of the most frequent instructions in the CPU and the performance of the computer system was synonymous with that of the processor. It was considered that the faster the addition instruction, the better the performance of the computer. So the addition instruction was used as the sole workload, and the addition time was accounted as the sole performance metric.

- **Instruction mixes:** With the arrival of new instruction for the processor, this metric did not reflect the performance of the CPU any more and workloads with more detailed instructions were required. A common approach was to measure the frequency of instructions during real execution. Then this usage frequency was coupled to the specification of various instructions. The result is the instruction mix that can indicate an average instruction time for a given mix by the timings of the different instructions. The averages can be used to compare different processors. One of the first examples of these instruction mixes is the Gibson mix [GIBSON, 59], which uses thirteen different instruction classes.

  In another example the performance of CPUs could be compared on the basis of their throughput. This can be done by the inverse of average instruction time, commonly quoted as the MIPS (Millions of Instructions Per Second) or MFLOPS (Millions of Floating-Point Operations Per Second) rates for the processor.

However it must be pointed out that

a) the instruction mixes only measure the speed of the processor. This may or may not have an effect on the total system performance when the system consists of many other components. System performance is limited by the performance of the bottleneck component, and unless the processor is the bottleneck (that is, the usage is mostly compute bound), the MIPS rate of the processor does not reflect the system performance.
b) It is meaningless to compare the MIPS of two different CPU architectures, such as Reduced Instruction Set Computers (RISCs)[17] and Complex Instruction Set Computers (CISCs)[18], since the instructions on the two computers are unequal. [JAIN, 91]
Another possible benchmark is the BOGOMIPS that gives, like MIPS, an indication of the processor speed and is quoted as "the only portable way over the various CPUs (Intel-type and non Intel-type) for getting an indication of the CPU speed" [DORST, 06]

However, today the number of instruction classes has risen enormously, but not all these classes can be reflected in the mixes. For example, the cache and pipeline technologies can greatly influence the performance. Also, it depends on parameter values like the frequency of zeros in matrixes or the frequency of a taken conditional branch.

In spite of these limitations, using instruction mixes can be interesting for comparisons between other computers of similar architectures or for estimating the execution time for key algorithms in application and system programs.

- **Kernels:** In brief, kernels are generalizations of instruction mixes and are workloads that consist of the most frequent function used by researchers. Different processors can then be compared on the basis of their performance on this kernel operation. Examples of kernels include Sieve, Puzzle, Tree Searching, Ackermann's Function, Matrix Inversion, and Sorting.

  But even kernels do not reflect the total system performance correctly because typically they do not take into account I/O devices.

- **Synthetic programs:** More and more applications use the I/O devices and they become an important part of the workload. Simple exerciser loops are used to measure the I/O performance. Therefore, the loops make a determined number of service calls or I/O requests and we can compute the average CPU time and elapsed time for each request or call.

  But not only I/O devices can be measured; also operating system operations such as process creation, forking, and memory allocation. Exerciser loops can be developed quickly and can run with non-real data files. The advantage is that the programs can be modified easily and ported to different systems.

[17] RISC is a microprocessor CPU design philosophy that favours a smaller and simpler set of instructions. Examples are ARM, DEC Alpha, PA-RISC, SPARC, MIPS, and IBM's PowerPC

[18] On the other hand, CISC is a microprocessor instruction set architecture in which each instruction can execute several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction. Examples are CDC 6600, System/360, VAX, PDP-11, Motorola 68000 family, and Intel and AMD x86 CPUs

Nevertheless, they are often too small and they do not make representative memory or disk references. In addition, the mechanisms of disk cache and page faults might not be adequately exercised. Especially for grid environment, the exercisers are not suitable because the loops may create synchronizations, which may have an important influence on performance.

So, if we apply this principle to some I/O components, we could measure with a special exerciser for example:

- the RAM read and write speed.
- the SWAP read and write speed.
- the disk read, write, and delete time in sequential or random mode,
- the network bandwidth (with different protocols like TCP or UDP - upload or download)
- the jitter (statistical dispersion in the delay of the packets)
- the network latency.

- **Application benchmarks:** If the benchmark is used to compare the performance of the machines for a particular application, the benchmark may consist of a representative subset of functions used in the application. In this way, the benchmark evaluates the performance of almost all resources in the system, including processors, I/O devices, networks, and databases. For example, the performance model can help to build such application benchmarks.

More examples about benchmarks can be found in the book [JAIN, 91].

We want to cite some important characteristics that will influence our choice for the design of a new information service:

1. Each benchmark can load the system in a critical way, so their first disadvantage is the limited application field in real time. For example a benchmark that tests the read and write performance of a hard disk will use the hard disk to its limits. This load is by no means at all desired at the execution time of disk sensitive applications.

2. Additionally, the execution of the benchmark takes a certain amount of time and if execution does not last long enough, the benchmark could be imprecise because spikes could be evaluated and not the average. But if the benchmark process takes too long, it would encumber the proper execution of the other applications.

3. Another problem is that the benchmarks are sometimes not generally accepted. And if we cannot compare our results to other results, the term evaluation does not make sense. For example, a first server can run the benchmark, but another server situated under another administration is not able to run it. Despite the excellent results of the first server's benchmark, we are not capable to compare the two servers on the basis of this benchmark.

Simple requests

We have seen how data are transported, processed, analysed and used to achieve better performance, but we not have seen all possibilities of how data are generated. Simple request is one type of data source that is instantaneously accessible and less intrusive. We have to distinguish between information that is *simple information* such as the number of processors actually in use or the CPU frequency and *historical information* that is already stored and resumed by the system without the explicit demand of applications. For example, the load average of the CPU and some statistics of the disk are instantaneously accessible via the operating system.

As the operating system is itself a monitor that stores an important amount of statistical information about the system, it is quite easy to use these data for other purposes like performance estimation. In this way, information is available about memory size, CPU information, memory and CPU usage, the network usage, the number of processes on the computer, and even statistics about the disk access. In addition, remember that performance does not necessary mean speed; the status of a process can also be interesting, for example, to know whether an application is running, killed, or sleeping. The sensor need only know the right directories where the information is stored, or the functions that provides the information.

Other sources can be the programming language that often provides the same information as the operating system, but in a more standardized way. These aspects will be discussed in detail in the next chapter.

4.4 Observations

As we have seen in the last chapter, the DIET scheduler can access information about the performance evaluation and performance prediction of resources. They are provided by three different tools:

- The CPU, the free memory, and the network performance between two hosts are provided by the resource performance forecasting tool NWS.
- A predicted execution time for the request is accessible via the application-centric performance prediction tool FAST.
- And finally the plug-in scheduler provides information internal to DIET, information like the last execution time of the SeD.

In the last chapters we have already pointed out some problems that consist of the current version of DIET with this set of data. We will summarise the different problems here and then propose some solutions.

1. Some servers would never have FAST fully installed. Either because FAST or NWS does not support the hardware architecture or because the services offered cannot be described accurately enough in the FAST language for a valid bench campaign or estimation

2. The installation and configuration is very difficult because it depends on several programs namely NWS, OPENLDAP, and GSL. These again depend on other programs like Cyrus SASL 2.1.18+, OpenSSL 0.9.7+, POSIX REGEX software, Sleepycat Berkeley DB 4.2+, or LTHREAD compatible thread package. Some of them are required, others are recommended, depending on the features one wants to install.

   Once installed, all these programs must be configured. For example, NWS with its – at least – three components (NWSNameserver, NWSSensor, NWSMemory) needs three configuration files. OPENLDAP is difficult to configure too, and FAST needs the information about the location of the other components and a description of the service written in the FAST language.

3. We know that the FAST predictions are based on the results of the benchmark database. These benchmarks are run at the time of installation of the services to the FAST component. For some services, these static measurements can incorrectly the real resource needs of the service represent.

4. We have seen that FAST is particularly suited to numerical algebra routines whose performance is not data-dependent and where a clear relationship exists between problem size and performance. Nevertheless if this relationship is not clear or there are other routines used, FAST's prediction would be not be accurate enough.

5. The decision to use DIET-external programs has advantages but also disadvantages: the created dependence of DIET on other software implies not only waiting time for bug correction, but also permanent code adaptation due to API changes in new releases.

6. FAST only[19] uses the information incoming from NWS, so there are a lot of parameters that are not taken in account (i.e. the read/write operation of the disk).

7. In principle, the default scheduling policy of DIET prioritises servers that are able to provide the performance prediction information of FAST and NWS. In general, this approach works well if all servers in the DIET hierarchy are able to provide these estimations. However, load imbalances may occur if the concerned platforms are composed of SeDs with varying capabilities: since DIET systematically prioritises server responses containing FAST and/or NWS data, servers that do not respond with such performance data will **never** be chosen.

8. Hence, there are many reasons to refuse the deployment of FAST in DIET. But if we do not use FAST, only NWS and the plug-in scheduler remains for accessing performance information. Only three performance related metrics[20] remain, and as in the critique above, the scheduler will not take into account all performance parameters that influence the execution of the request. Additionally, NWS is only accessible via the interface of FAST, so if FAST is not installed at all, NWS is unavailable too. Finally the scheduler will only work with a round-robin mechanism.

[19] FAST can also use its own, basic sensors, but they are not complementary to the information provided by NWS

[20] NWS provides much more than two kind of information (i.e. CPU frequency, CPU number, memory speed, file read and write speed, bandwidth), but there are not yet access functions in DIET implemented, or they are not public (the bandwidth measure is only available for DIET programmers)

9. Another problem consists of the way we can access these metrics. The plugin scheduler has already defined one function for the access on FAST- and NWS-based metrics and another function for the last execution time. At the moment this solution works fine, but what happens when new performance metrics will be available? It would be necessary to access each metric by a specific function.

4.5 Summary

We have seen in this chapter the general data dependency of the different tools: The sensors that provide performance data from different devices to the monitors, the monitors that offer the information to the performance prediction tools for the resources and the latter provide this forecast to the application-specific performance prediction tool. With the example of FAST and NWS we have seen that very complete and precise solutions are already available, but unfortunately, not all aspects of the performance prediction are covered, or the solution is too complex for a simple use. So we need an intermediate solution between highly complex prediction and nothing at all. If we recapitulate,

- we need metrics even if the environment is very heterogeneous,
- we need a simple tool for basic metrics in the worst case scenario(i.e. if no other metrics are available),
- we need more metrics covering more aspects of performance,
- we need a standard access to the different metrics.

The implementation of these requirements can be read in Chapter 5.

Chapter 5

CoRI

The analysis in the previous chapters shows an important need for measurement tools. As we have seen in the last chapter, DIET needs reliable resource information for scheduling provided by grid resource information services. Our job here is to take up the challenges cited at the end of the last chapter and to find an adequate solution for DIET.

In this chapter we will introduce the exact requirements of DIET for a grid information service, the architecture of the new tool CoRI, (Collectors of Resource Information), the different components inside of CoRI, and the problems that we have encountered during the different developing phases.

5.1 Requirements analysis

The requirement analysis includes functional requirements and non functional requirements. In the following sections these two types are elucidated.

Functional requirements

As we have seen in the last chapters, the scheduler is one of the elements of a grid infrastructure that can influence heavily the performance of the system. We have seen that the scheduler can only work correctly if it receives adequate information on the part of grid information services. Although DIET has already a working performance prediction tool (namely FAST), it is necessary to create a complementary information services. The reasons are cited in Section 4.4. The following functionalities should be implemented in the new tool:

- The tool has to provide a basic set of performance measurements that can satisfy basic scheduler needs.
- The information has to be stored in estimation vectors (explained in Section 5.2.2).
- The tool must always provide an answer in order to avoid the blockage of the grid system. If the tool is not able to provide a measurement, a generic response must be provided.

In addition, the system (or the developer-defined scheduler) must be able to distinguish between real responses and those generic responses.

- The tool must provide one single interface for all kind of resource information services.

Non-Functional requirements

- **Ubiquity**: Because a grid is deployed in heterogeneous environments (as we defined the characteristics in Chapter 1), the tool must support the resulting various operating systems, the different hardware infrastructure and the other difficulties due to the heterogeneity.
- **Extensibility**: The tool should not only provide adequate information for the existing DIET (that is why we have made the state of art in the second chapter), but should be extensible for new measurements needed in the future. Additionally, the tool should be extensible for the use of other programs like Ganglia, RMI or REMOS.
- **Accuracy and latency**: The tool must provide accurate performance measurements in a **timely manner**.
- **Non-intrusiveness**: The tool must load the resources for its measures as little as possible.
- **KISS**: Since DIET offers already complex performance prediction services, the demanded tool should follow the approach "Keep It Simple and Stupid".
- **Concurrence**: Because different services run concurrently on the grid resources, the tool must support concurrently access to measurements.
- **Invisible**: The tool should not hinder the execution of grid components in any way.

5.2 Solution

The new tool has to solve two main problems: firstly, it must provide basic measurements that are available no matter the context of the system. The service developer can rely on this **collector of resource information** even if no other resource service like FAST or NWS is installed. Secondly, the tool must **manage** the use of different collectors at the same time and in a similar way.

As there are two problems, we offer now two solutions: the **CoRI-Easy** collector for the first problem, namely the collector, and the **CoRI Manager** for the second problem, namely management of different collectors. In general, we will name both these solutions together the **CoRI** tool, which stands for Collectors of Resource Information. Both solutions are described in the following section.

5.2.1 Global architecture

The CoRI-Easy is a set of simple requests for basic resource information, and CoRI Manager will allow the developer team to add other resource information services. As CoRI-Easy is a resource information service, it seems to be logical to add it as collector to the new CoRI Manager. Figure 5.1 shows the architecture of CoRI. We can see that not only CoRI-Easy is available as collector but FAST as well. In addition, this figure shows that it is still possible to add new collectors (indicated here by "other collectors").

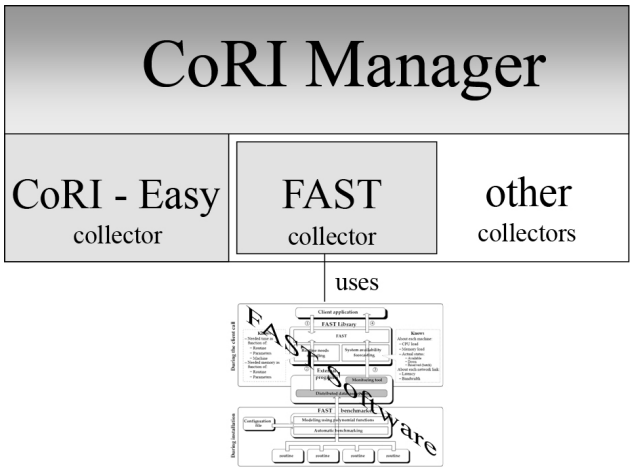

Figure 5.1: The architecture of CoRI: The CoRI-Manager and its collectors, namely the CoRI-Easy and the FAST collector.

We will discuss the CoRI Manager first, and then the CoRI-Easy module.

5.2.2 CoRI Manager

The CoRI Manager allows the access to the different **modules** (also referred to as **collectors**). A module is any kind of element that can provide information about the system. This **modularity** allows the separation of measurement sources and the selection of to each module. Even if the manager should unify the different resource information services, the trace of data remains, and so the origin can be determined. For example it could be important to distinguish the data coming from the CoRI-Easy module and the FAST module, because the information of FAST would give a better estimation of the real value. The **extensibility** is ensured by the modular design too. Because the interface of the manager allows in some steps to add a new module, additional modules like Ganglia or NWS are possible.

We will first introduce the structure of the metrics used in DIET for the exchange of performance data, and then present the different CoRI Manager functions.

Estimation Metric Vector

The following vector is described in DIET User's Manual [BOLZE and al., 06], and is reused here for CoRI. This dissertation has not contributed to its design or implementation work.

In DIET, information services on SeD-side store their performance measurements in *estimation vectors*, and the agents and the service developer can access this data for scheduling purposes.

The vector is divided into two parts, the first part represents standard performance measures that are available through CoRI (for example, the number of CPUs, the memory usage,...) and the plugin-scheduler facility (i.e. the last execution time), and the second part is reserved for developer-defined measurements that are meaningful solely in the context of the application being developed.

The vector supports storage of single and multiple values, because some performance prediction measurements are not only single values (called scalars) but a list of values (i.e. the frequency of each processor). As there are different performance measurements, the estimation vector does not store only one type but multiple types. So the estimation vector is a container for multiple lists and multiple scalars.

Figure 5.2 is an estimation vector sample that uses different measurement types (identified by tags, see next section), namely the number of CPUs, the frequency of the processors and two developer-defined measurements. We can see that the vector indicates that this component has two CPUs and their frequencies are 20 and 50 MHz respectively. The lower part of the estimation vector represents metrics defined by service developer. In the figure, the numbers could represent 25 tasks running, a first file is available (1), and a second file is not available (0)

tag	scalar	0	1
EST_NBCPU	2		
EST_CPUSPEED		1400	2000
0	25		
1		1	0

Figure 5.2: An example of the estimation vector with some estimation tags.

In the current implementation of DIET, estimation vectors are not stored persistently, so they are lost when the request is scheduled and cannot be shared with other requests or DIET services.

Estimation Tags

Since the vector contains different information types, each standard performance measurement has its own tag in the vector. This tag allows direct access to the measurement. Therefore, the Table 5.1

describes the correspondence between measurement and tag.

Additionally, developer-defined performance measures can be stored in the vector and use integers for identification.

In the Table 5.1, the first column indicates the name of the tag, the second column marks the measurements that provides a list of values instead of a single value, the third column gives a short explanation of the provided value (with its unit of measurement).

In addition to these tags, we have a special tag, namely `EST_ALLINFOS` that is an empty record in the vector, but can be used to demand all known fields of a particular collector.

`Information tag` `starts with EST_`	multi-value	`Explication`
TCOMP		the predicted time to solve a problem
TIMESINCELASTSOLVE		time since last solve has been made (sec)
FREECPU		amount of free CPU between 0 and 1
FREEMEM		amount of free memory (Mb)
NBCPU		number of available processors
CPUSPEED	x	frequency of CPUs (MHz)
TOTALMEM		total memory size (Mb)
BOGOMIPS	x	the BogoMips
CACHECPU	x	cache size CPUs (Kb)
TOTALSIZEDISK		size of the partition (Mb)
FREESIZEDISK		amount of free place on partition (Mb)
DISKACCESREAD		average time to read on disk (Mb/sec)
DISKACCESWRITE		average time to write to disk (sec)
ALLINFOS	x	[empty] fill all possible fields

Table 5.1: Explanation of the estimation tags

API of CoRI Manager

The interface of the CoRI Manager is available for DIET service developer (and implicitly also available for DIET developer as well) and consists of three functions. The first function allows the initialisation of the indicated collector and adds the collector to the set of collectors which are under the control of the CoRI Manager.

```
int
diet_estimate_cori_add_collector(diet_est_collect_tag_t collector_type,
                                 void* data);
```

The `collector_type` is used to designate a particular collector. In this way, the user is in full control of measurement sources, so he can decide which collector should be used for measuring. In

fact, only `EST_COLL_EASY` and `EST_COLL_FAST` are accepted, since the CoRI-Easy and the FAST modules are the only collectors implemented yet. The second parameter, namely `void* data`, contains specific information for each collector. Actually, neither FAST nor CoRI-Easy need additional information, but as we wanted to create an extensible solution, we have already anticipated this case.

The second function is the access to measurements:

```
int diet_estimate_cori(estVector_t ev,
                       int info_type,
                       diet_est_collect_tag_t collector_type,
                       void* data);
```

The first parameter is `ev` the estimation vector, which is used to store the measurement. The second parameter is the tag which we have seen in Section 5.2.2 and which specifies the type of measurement (i.e. the number of CPUs, the CPU load average,...). The user must also specify which collector should be used to gather the information, so the `collector_type` must be indicated. The `data` parameter is used for the same reason as for the first function: sometimes additional parameters are necessary for the particular measure. FAST needs the description of the problem (also called `profile`) for creating its application-specific performance predictions.

CoRI needs additional information for

- CPU load average (it must be indicated whether the average is computed based on 1, 5 or 15 minutes),
- for disk speed benchmarks (the path where the benchmark should be executed must be indicated),
- for partition size measure (a path of a directory of the partition must be indicated)

But if CoRI-Easy does not receive this information, it uses default parameters (i.e. 15 minutes average for the CPU load and the current director for the disk benchmark and the disk size measure).

The last function is available to test the availability of CoRI-Easy.

```
void diet_estimate_coriEasy_print();
```

The result of this function can be similar to the following print screen:

```
start printing CoRI values..
CPU 0 cache : 1024 Kb
number of processors : 1
CPU 0 BogoMips : 5554.17
cpu average load : 0.56
free cpu : 0.2
disk speed in reading : 9.66665 Mbyte/s
disks peed in writing : 3.38776 Mbyte/s
total disk size : 7875.51 Mb
available disk size  :373.727 Mb
total memory : 1011.86 Mb
available memory : 22.5195 Mb
end printing CoRI values
```

5.2.3 CoRI-Easy module

The CoRI-Easy module is a resource collector that will provide basic performance measurements of the SeD. Due to the interface with the CoRI Manager, the service developer and the DIET developer have the possibility to access to CoRI-Easy metrics. We first introduce the design solution of CoRI-Easy, then its API and finally we will discuss some specific problems. CoRI-Easy should be extensible like CoRI Manager, i.e. the actual functions must be easily replaced, extended or completed by new functions. So it is important to choose an appropriate architecture that response to these requirements. Figure 5.3 shows two possible designs for CoRI-Easy which we will detail now.

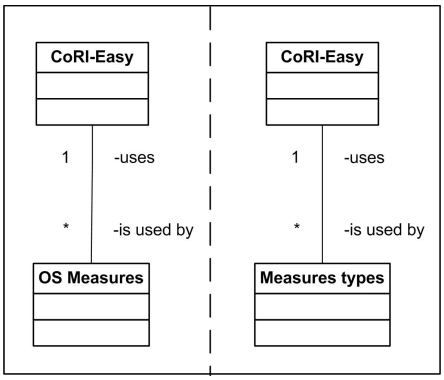

Figure 5.3: Two possible architectures: Either sorting by operating systems, or by measure types

- If we use an operating system approach, we will first identify the operating system type, and then call system functions provided by the operating system or "independent" functions (i.e. functions that runs on multi-platform, or multi-environments). If we want to support a new operating system, we add in the CoRI-Easy hierarchy a new class `C++` that contains all measure functions. However, this solution has certain disadvantages, e.g. the question of how we can identify the operating system. There is no standard function for accessing this information. There are also design weaknesses because some functions are used for all operating systems (like `loadaverage`), so there will be no real line between the different operating systems. Furthermore, a lot of UNIX derivations are actual used[1] and each operating system would need its own set of measure functions. Hence, the work for analysing the operating system, finding out the functions and integrating them in CoRI-Easy would be enormous and so we will not choose this architecture.

- On the other hand we can use a functional approach. Therefore, we categorize the functions by the information they provide. So each information type has its own class. For exam-

[1] Indeed, since the first UNIX version, the operating system has evolved and many derivations are available (an example of the UNIX history is available at `http://www.levenez.com/unix/history.html#10`)

ple every function concerning information about the CPU will be implemented in the class `Cori_easy_CPU`. This avoids to the need to find out which is the used operating system. We must simply test the existence of the function, so there is even the possibility to find some information on unknown operating systems. So in this approach, it would be easier to add a new measure type to the existing types (namely the CPU, memory and disk performance measures). However, if we accept to test each function dynamically, instead of assuring the availability on each operating system, we would have no real overview of the supported operating systems. In addition, some measurements are not easy to classify, for example the bus speed between RAM and CPU. Does it belong to the CPU class, to the RAM class or to a new class? Despite of these disadvantages this is the approach we have chosen for CoRI-Easy.

So each `C++` class contains functions for a specific information type. Actually, the class `Cori_Easy_Disk` is responsible for all measures concerning the hard disk. And the same methodology is applied to the classes `Cori_Easy_CPU` and `Cori_Easy_Memory`. So if DIET developer wants to insert network measures, he must create a `Cori_Easy_Network` and implements the measure functions for the network.

The second mechanism allows the DIET developer to define the hierarchy of function calls. The hierarchy is necessary because sometimes different ways are possible to deliver the measurement. And each way has its advantages, but as well its disadvantages. So it is up to the DIET developer to decide the right way. This problem will be discussed in detail in Section 5.2.3.

Our goal was not to create another sensor system or monitor service, so Cori-Easy is a set of basic system calls for providing basic performance metrics. Furthermore the metrics are available via the interface of the CoRI Manger.

But more important than the question "how do we measure?" is the question "what do we measure?". This question is the concern of the next section.

Provided measurements by CoRI-Easy

As we have seen in Section 4.2, many elements influence the performance, so it would be an enormous deal of work to take into account all this information. For this reason, we will select and discuss different measurements that are more important for performance evaluation. Due to the state of art presented in Chapter 4, we were able to extract a set of information that seems crucial for a basic scheduling. Of course, each scheduling policy is different, so it would be contradictorily to say that this set is indispensable and complete.

In addition, the simplicity and non intrusiveness of the function have influenced the choice of metrics. As the function can be called simultaneously many times, and because other tools are already available for DIET, the following information can be provided under these conditions.

CPU power CPU power and CPU availability are in many articles the most important factor for performance[2]. Therefore we provide different measures for CPU performance evaluation. First, CoRI-Easy provides **hardware information** namely the *number of CPUs*, the *CPU*

[2]But as we have seen in Chapter 4, the performance depends not always on the CPU

frequency and the *CPU cache size*. These measurements do not give great indication about actual load of CPUs, so we added more **situation related information**, namely the *BogoMips* for indicating the CPU power and the *load average* and *CPU usage* for indicating the CPU charge.

The BogoMips indicates in a special way the speed of the processor by measuring how fast a certain kind of busy loop runs on a computer. Irrespective of whether this kind of measure is too unscientific, we provide it here, because it seems to be one simple and portable way to compare two processors approximately.

The CPU usage indicates the actual CPU load. The fraction includes all CPUs actually available on the machine. This measure represents an actual state of the machine and allows schedulers to be high reactive.

That is why it is suitable for high rate scheduling, i.e. when many small requests must be scheduled at the same time, it is important to know the instant load of the different SeDs.

The load average indicates the average CPU queue length of runnable processes. So if the value is equal to 0, no process using the CPU and a new process can benefit from the entire CPU resources. When the load average is equal to 1, a mono processor machine would be fully loaded, but a dual processor has still resources available. With a load average of 4 the mono processor should be a quad processor for treating all processes in time.

This measure represents a long term CPU load measurement. It can be used for any size of requests, but it is not suitable for high rate scheduling executed in short time due to a low reactivity.

Memory capacity The memory is the second important factor that influences performance. The provided metrics are the *total memory size* and the *available memory size*.

Disk Performance and capacity Finally, performance and capacity of a device can be measured. CoRI-Easy provides metrics about the *read and write performance* of any writable device, as well the *maximal capacity* and the *free capacity* of any device.

Network performance CoRI-Easy should provide some indications about the performance like the latency and error transmission, but functions for these metrics were not elaborated for time reasons.

Sources of information

Now that we have chosen the metrics for CoRI-Easy, we must decide how to extract this information. This section will explain the problems that arise if we try to extract low-level information from heterogeneous systems.

1. Some operating systems provide functions for gathering information like the CPU frequency, the cache size, ... but in heterogeneous systems, the functions may not be available for each operating system.

For example, some Linux-based operating systems provide the virtual directory `/proc` for information about the CPU, but the MacOS operating system does not provide this directory.

Some functions are only available since a certain version. For example, the repertory `/proc/diskstats` for disk statistics is only available since Linux 2.5.69.

2. All roads lead to Rome. In our context this would mean that a lot of functions are available for gathering the information. But the semantic or syntax of these functions are rarely identical on the same operating system or on different operating systems. For example, the CPU load average can be read with different methods: by the virtual directory `/proc`, by the GNU C library with the function `getloadavg`, or by the command `top`. However, not each function provides the same accuracy. The command `top` has in general a refresh rate of three second, but no standard specifies that it is the same on each Linux distribution! On the other hand, we have the GNU C library, where the function is specified, accepted and standardized. So finally, which of the two functions would we choose now for the measurement?

   Another example is the interval of average for the load average: "On some systems, such as Digital Unix operating systems, these are 5, 30, and 60 second averages, while on others, such as Linux, these are 1, 5, and 15 minute averages" [DINDA and al., 99].

   A last example: The command `top` support the parameter `-n`, but the call `top -n 1` on RedHat would refresh the measurements one times, contrary to the operating system FreeBSD which would display only the first process in the list.

3. The deployment of a grid architecture should not require special authorization for its execution. Unfortunately, some functions provided by the operating system need advanced privileges for the call. For example, only members of the group "disk" can execute the command line program `hdparm`.

We have analysed the different sources where the information can be retrieved and we can point out now different source levels. Each level represents a different degree of portability and the higher the level, the higher the portability. This allows us to define our preferences. This means if at the highest level a function provides a metric, we will advantage this function.

- The **standard C library** is the highest level because DIET code is based on standard `C`, so we are sure to have at least the functions provided by this level for our measure functions. Indeed some of them are written in simple `C` code without external system calls, for example the benchmarks for reading and writing speed of devices.
- The **GNU Library C** is an extension of the standard C library, so different additional functions are available. It is used by the program OmniORB[3] which is in turn in DIET. So we are assured that we have this set of information too. Amongst others, the function `int get_nprocs (void)` that gives the number of processors, use this library.

[3] omniORB is a robust high performance CORBA ORB for C++: `http://omniorb.sourceforge.net/index.html`

- The third level is the **POSIX standard**[4] . According to The Open Group, one intended group of people for these standard are "Persons developing applications where portability is an objective" [POSIX, 04]. So, it should be suitable for our purpose and we use this standard for information about memory via the command line program `ps`, or information about the disk via the command line program `df`. Nevertheless the status as standard, not every operating system developer must follow this standard, so it is not sure that all operating systems provide these functions. In addition, the set of functions defined by the POSIX standard is restricted and therefore other measures functions must be found elsewhere.

- The fourth level is the **operating system** but we have already seen that it would be very difficult to support all systems. In this category we use, for example, the virtual directory `/proc` for information about the processors (frequency, cache, BogoMips[5]).

- So, if the operating system makes problem, we could perhaps avoid the operating system and attack directly the lower level. . We have therefore analysed the Linux kernel code for extracting the function that provides the number of processors. Unfortunately, the function consists of a set of other functions where each of these functions is responsible for one specific processor type. It would be an enormous work to extract these functions, and to maintain the set of function for new processor types. Moreover, one principle of the operating system is to hide such technical details, so we should abide by this rationale.

So actually, we work with four different levels, namely the C, the GNU, the POSIX standards and the operating systems. As we have seen now, multiple levels can be used for measure functions. This is why CoRI-Easy functions utilize a set of other functions that are tested for their availability. If the first function doesn't work, the next one is tested.

Furthermore, we have to guarantee low latency. As the most of these functions use virtual directories access or standard functions, we should be able to provide fast response times. In order to show the performance of these calls, we have made some tests for the different measures. The following test is made on a Pentium 4, 2.4 GHz, 512 Mb RAM, with the operating system Linux 2.6.16. We will loop the calls of each measure during 5 seconds.

Table 5.2 represents the benchmark tests and we can see that some functions are very fast, and others too slow. This difference comes from the dissimilar functions we use: accessing the `/proc` directory takes less times than running the command program `df`. Also, we detect inefficient measures for reading and writing disk speed. These problems are due to disk technology: when the benchmark will write to the disk, the data will be written to the cache first and only then to disk. So we must write more than the disk cache to the disk to avoid the cache. Nevertheless, this approach raises some questions about the benchmark validity and generality, because the cache is part of the disk, and that is why its acceleration should be taken into account for measuring performance. Finally, we must reduce the utilization of these functions to maintain a reasonable latency for the system.

[4]The Portable Operating System Interface (POSIX) standardization are written and maintained by the PASC (Portable Applications Standards Committee)

[5]Actually, the BogoMips is a static value because it is measured only once at machine boot time.

Information tag starts with EST_	number of loops
FREECPU	242
FREEMEM	167 165
NBCPU	59 641
CPUSPEED	80 940
TOTALMEM	174 778
BOGOMIPS	88 136
CACHECPU	96 377
TOTALSIZEDISK	328
FREESIZEDISK	341
DISKACCESREAD	1
DISKACCESWRITE	36

Table 5.2: The low latency test for the CoRI measure functions

FAST module

We have already mentioned that it is possible to receive FAST performance prediction via CoRI Manager. The FAST collector is fully integrated into CoRI and the old API of FAST is deleted. One improvement is that the SeD side of FAST will be launched on explicitly service developer demand only.

5.3 Changes in the DIET software

Since CoRI is fully integrated in DIET, some changes of DIET code were necessary. In order to control these changes, we use a compiler's pre-processing mechanism. In this way we are able to deactivate CoRI and its impact on DIET code. So, it is still possible to use DIET without CoRI, but some important additional measurements are not available in this case. Table 5.3 shows the different compiler options and the measurements that are available with each compiler mode. For example if we compile DIET with FAST but without CoRI, we have the same information set as before implementing CoRI. However, this set was not exhaustive and for user that want to write their plugin scheduler, it would be interesting - but not mandatory - to compile with CoRI in order to have a full grid resource information service.

But if we compile DIET with CoRI, there are not only more measurements available, furthermore we made the following changes in DIET:

- The default scheduling has changed: we have seen in Section 3.5 the sequence for scheduling policy was based on FAST prediction. In case of non availability it was based on NWS resource forecasting, then a round-robin scheduling was performed, and finally, with no information,

	no cori		**--enable-cori**	
`Information tag starts with EST_`	no FAST	`-with-FAST`	no FAST	`-with-FAST`
TCOMP		x		
FREECPU		x	x	x
FREEMEM		x	x	x
NBCPU		x	x	x
CPUSPEED			x	x
TOTALMEM			x	x
BOGOMIPS			x	x
CACHECPU			x	x
TOTALSIZEDISK			x	x
FREESIZEDISK			x	x
DISKACCESREAD			x	x
DISKACCESWRITE			x	x
ALLINFOS			x	x

Table 5.3: This table indicates the set of available information depending on compile options

the random scheduling was performed. Now we have changed this sequence with the following reasons:

First, we have seen that the scheduling for NWS is rarely adequate. And second, as the plugin scheduler is available, it should be up to the developer of the service to integrate an adequate scheduling.

So, we have decided to remove the first and second scheduling policy, namely the FAST and NWS based scheduling.

- At the beginning of the CoRI project, the first requirement was to provide the standard estimation records of the estimation vector with measurements that would allow a default scheduling more appropriated than the round-robin. But as we have now seen in this dissertation, a default scheduling, which would be advantageous for any kind of program, is not possible, and an approximation is difficult to elaborate. That is why the search for a new scheduling has been suspended. And if no scheduler uses the CoRI measurements, it is not necessary to measure with CoRI on each request. So, the only change that was planned for DIET, namely the CoRI call for providing a full filled estimation vector, has not been introduced, and CoRI is used for developer-defined schedulers only. In the following chapter we will introduce some basic scheduler policies, but this is only the beginning for further researches.

5.4 Examples

In general we have tested the CoRI Manager on different operating systems like RedHat, Ubuntu, MacOS and FreeBSD. CoRI-Easy is successfully tested on the GRID'5000.

We provide a new example for the DIET project called CoRI example. This service is the computation of the Fibonacci number[6] with capacity until 2^{24} i.e Fibonacci of 46. This computation is very CPU intensive, but otherwise less non-CPU-resource demanding.

We will now testing small examples on the gdsdmi cluster.

5.5 Testing on the cluster GDS/DMI

The GDS/DMI is a cluster site, including

- 8 servers: SuperMicro 5013-GM, motherboard SuperMicro P4SGE, processor P4 2.4GHz, FSB400, 256Mo EGG RAM, 40Go hard disk, operating system Linux 2.6.16.
- 5 servers: SuperMicro 6013PI, motherboard SuperMicro X5DPR-IG2, processor Intel P4 XEON 2.4GHz, FSB533, 1Go RAM, 40Go hard disk, operating system Linux 2.6.16.
- 14 other servers.

We sent 140 requests on DIET architecture by a single client. We have used the plug-in scheduler for creating our own scheduling policy. This policy prefers the SeD with the lowest load average. As comparison we use the standard scheduling, namely the round robin.

5.5.1 Example 1: Simple request

In our first example, we submitted the requests on DIET architecture with 1 MA (MA_0), 2 LA (LA_0-LA_1) and 16 SeDs. By using the DIET tool VizDIET, we are able to visualize this architecture in Figure 5.4.

In this first example we launched the request in a time interval of 5 seconds. The tasks are identical workloads with an average solve time of 24 seconds. The total execution time of all requests is 3 363 seconds for the load average scheduler and 3 581 seconds for the round robin scheduler.

In this context, the new scheduler is nearly equal to the standard round robin scheduler. The Figure 5.6 and Figure 5.5 show the deployment of the request respectively for a load average scheduling and a round robin scheduling. Each request is represented is one block on the figure. Depending on the request's computing start time the figure indicates the SeD that computes the request and the computing time interval. We can remark that in Figure 5.6 2 pairs of task were executed on the same SeD at the same time. At first sight this should not be optimal, because there are at any time

[6] Fibonacci numbers form a sequence defined recursively by:

$$F_n := F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

other SeD that are not occupied. But as we work here on a heterogeneous system, the computing takes less time, and the load average is not varying so much. Indeed, the two computers 1 and 8 are used two times one consecutively. That is why their load average values rise and they are used more rarely. In this way more requests are attributed to the more powerful machines.

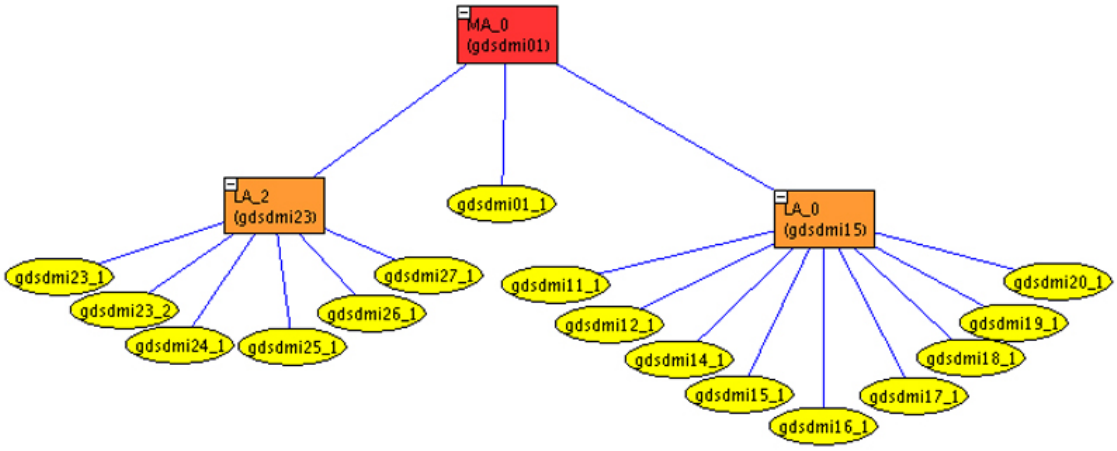

Figure 5.4: Example of a deployed DIET architecture visualized with VizDIET: The master agent, the local agents and the server daemons

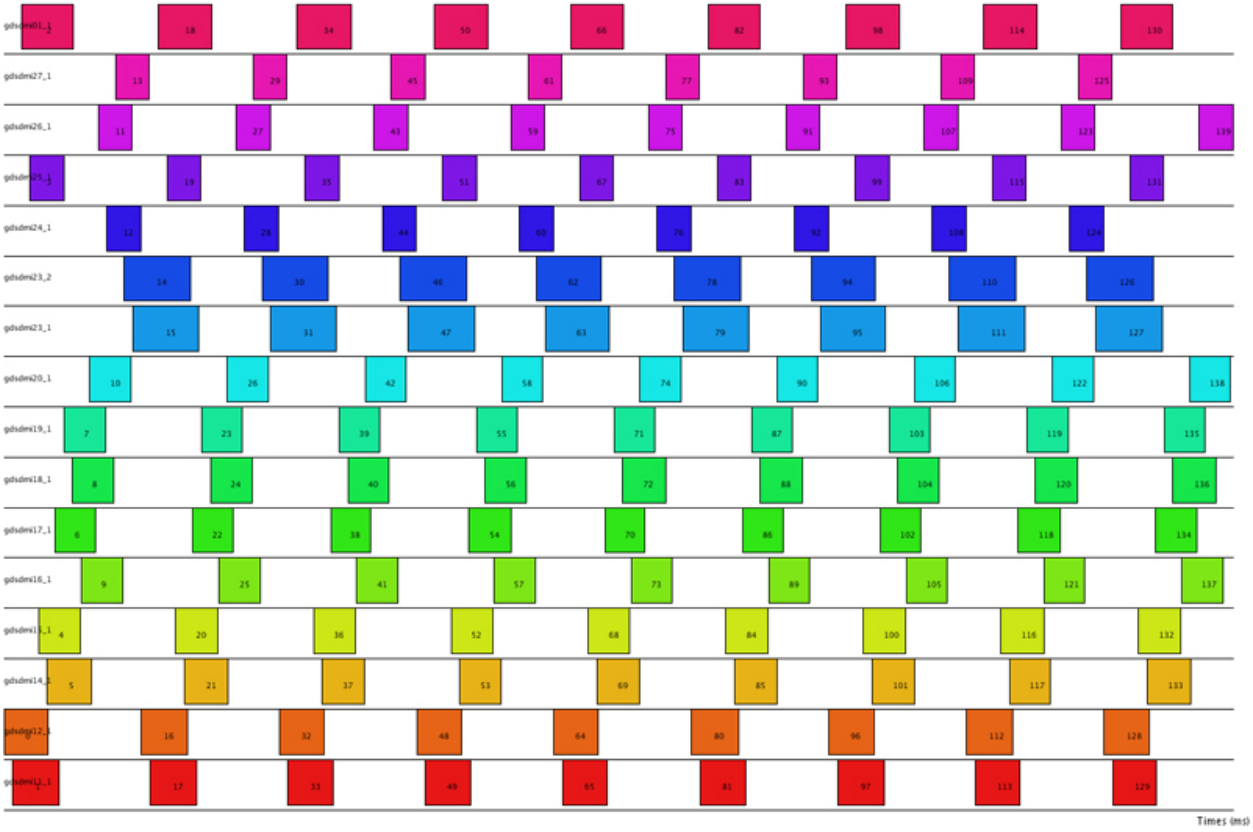

Figure 5.5: The Gantt view of scheduled requests on different SeDs for the test 1, round robin

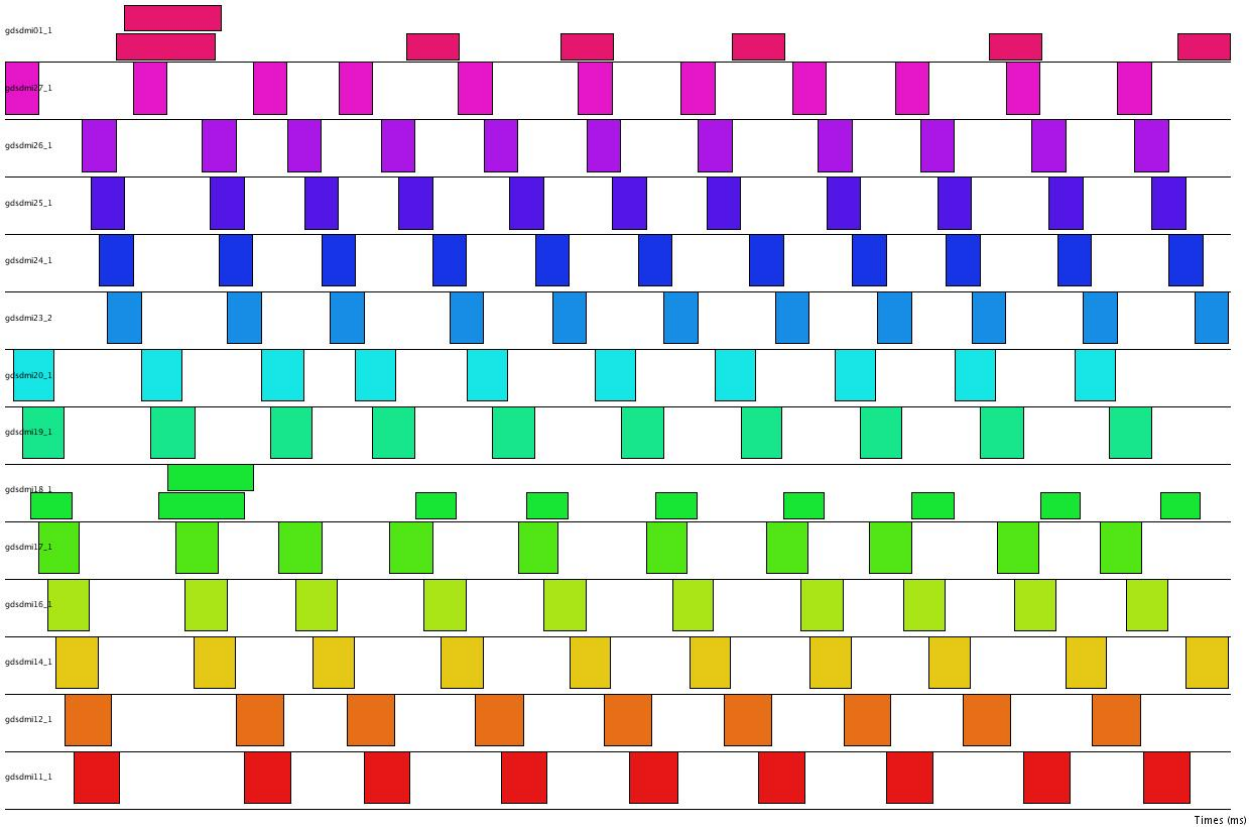

Figure 5.6: The Gantt view of deployed requests on different SeDs for the test 1, load average

5.5.2 Test 2

In the second and third example, we submitted the requests (of priority 22) on DIET architecture with 1 MA (MA_0), 4 LA (LA_0-LA_1) and 24 SeDs. Figure 5.7 visualizes this architecture. Moreover, we used a test workload for simulating external influence by other users. These processes are of priority 25 and generate a load average of 1 on the SeD gdsdmi14 until gdsdmi20.

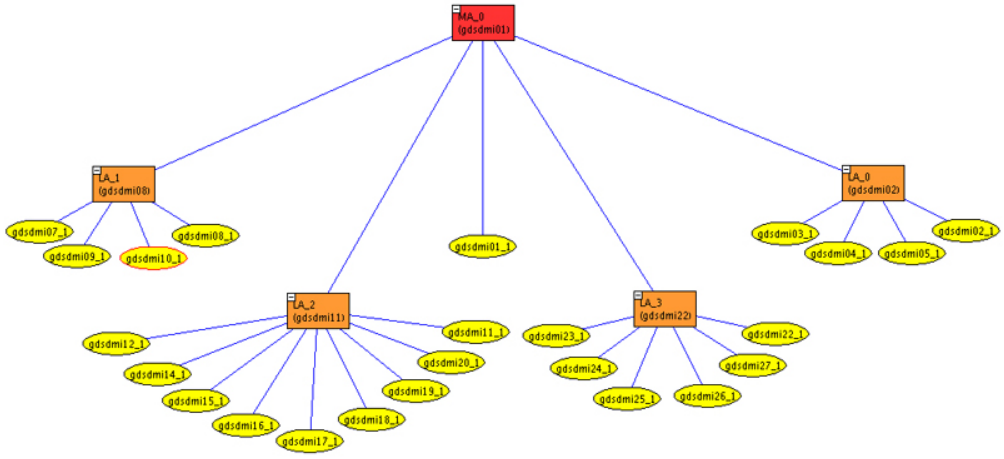

Figure 5.7: Example of a deployed DIET architecture visualized with VizDIET: The master agent, the local agents and the server daemons

So in the second test, identical tasks are submitted to the testbed at intervals of 1 second. The total execution time of all requests is 20 937 seconds for the load average scheduler and 6 992 seconds for the round robin scheduler.

The round robin scheduling is consequently much more suitable than the load average scheduling in this case. It is due to the low reactivity that too many requests are already sent to the lower occupied SeDs. These SeDs will be overwhelmed by the new tasks, and their low load average does not respond to the actual high load. The load average cannot replace the round robin scheduling for many big requests in short time intervals, but with enough knowledge about the request, it should be able to calibrate the load average for its purposes.

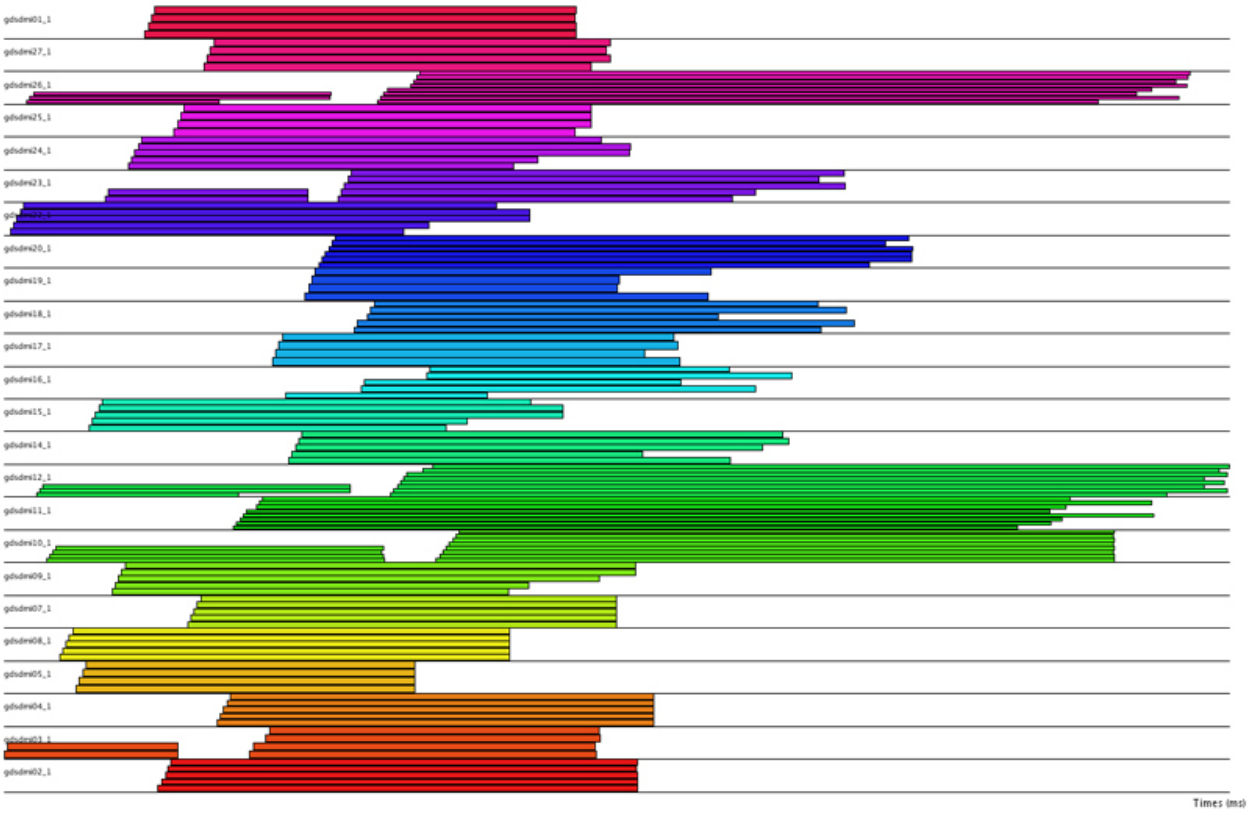

Figure 5.8: The Gantt view of deployed requests on different SeDs for the test 2, load average

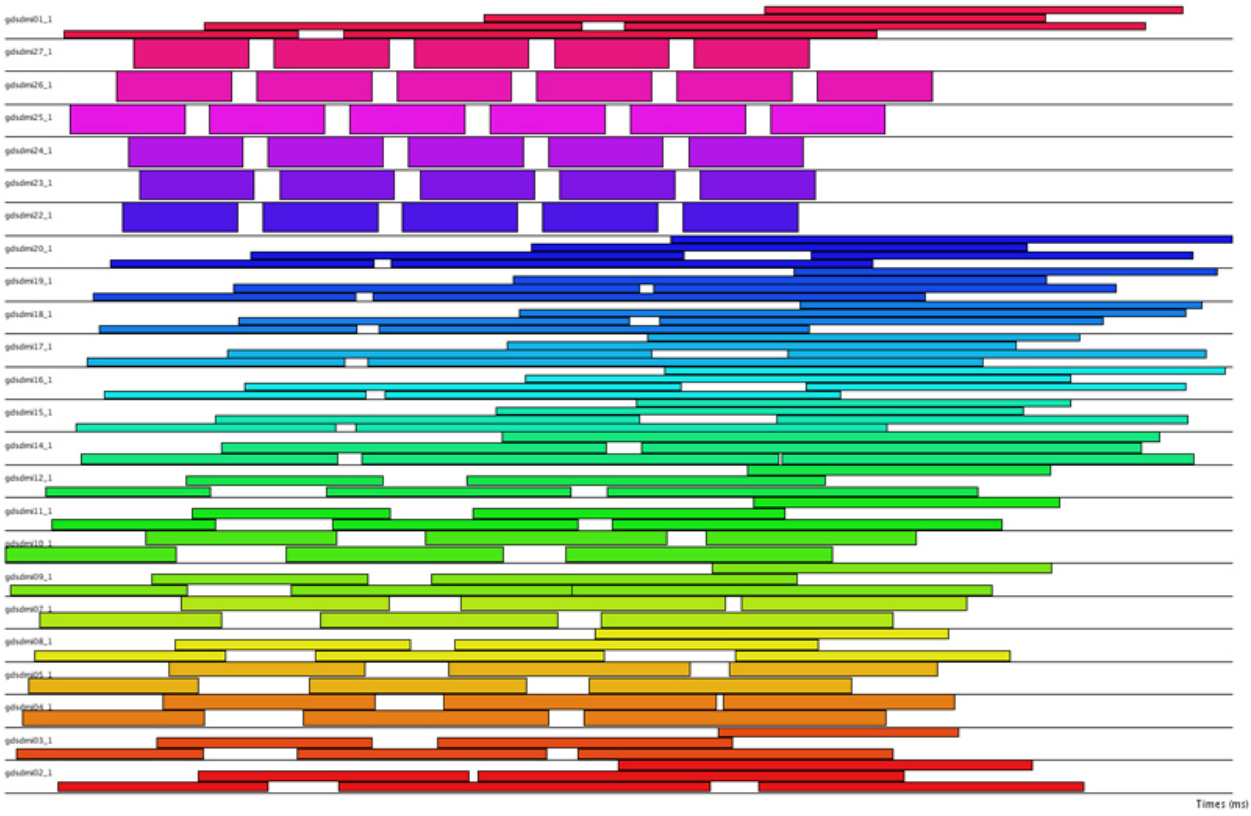

Figure 5.9: The Gantt view of deployed requests on different SeDs for the test 2, round robin scheduling

5.5.3 Test 3

As in test 2, we submitted 140 requests on a DIET architecture with 1 MA, 4 LA and 24 SeDs. The scheduling includes different workloads with different time intervals between the two requests. There are still test workloads on the machines gdsdmi14 until gdsdmi20.

The total execution time of all requests is 7555 seconds for the load average scheduler and 3024 seconds for the round robin scheduler. This bad execution time of the load average is again caused by the low reactivity. In the actual case, the requests have different computation time, and most of them are computed before the load averages of the computing SeDs exceed 1. Figure 5.10 shows this behaviour: Three of the 24 SeD do not have compute any request. Figure 5.11 shows that the round robin scheduling is more adapted for these request suite, but that there are still performance losses (requests are computed by already occupied server whether other free SeDs are available).

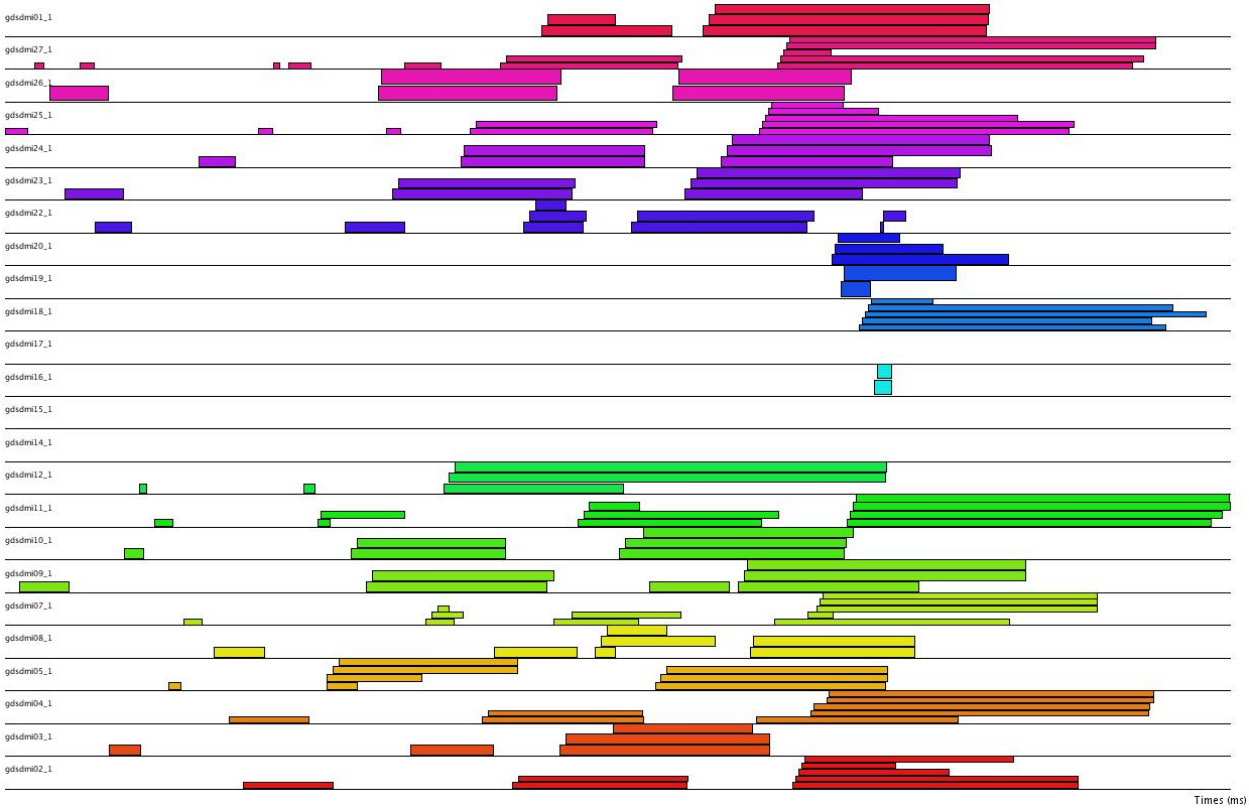

Figure 5.10: The Gantt view of deployed requests on different SeDs for the test 3, load average

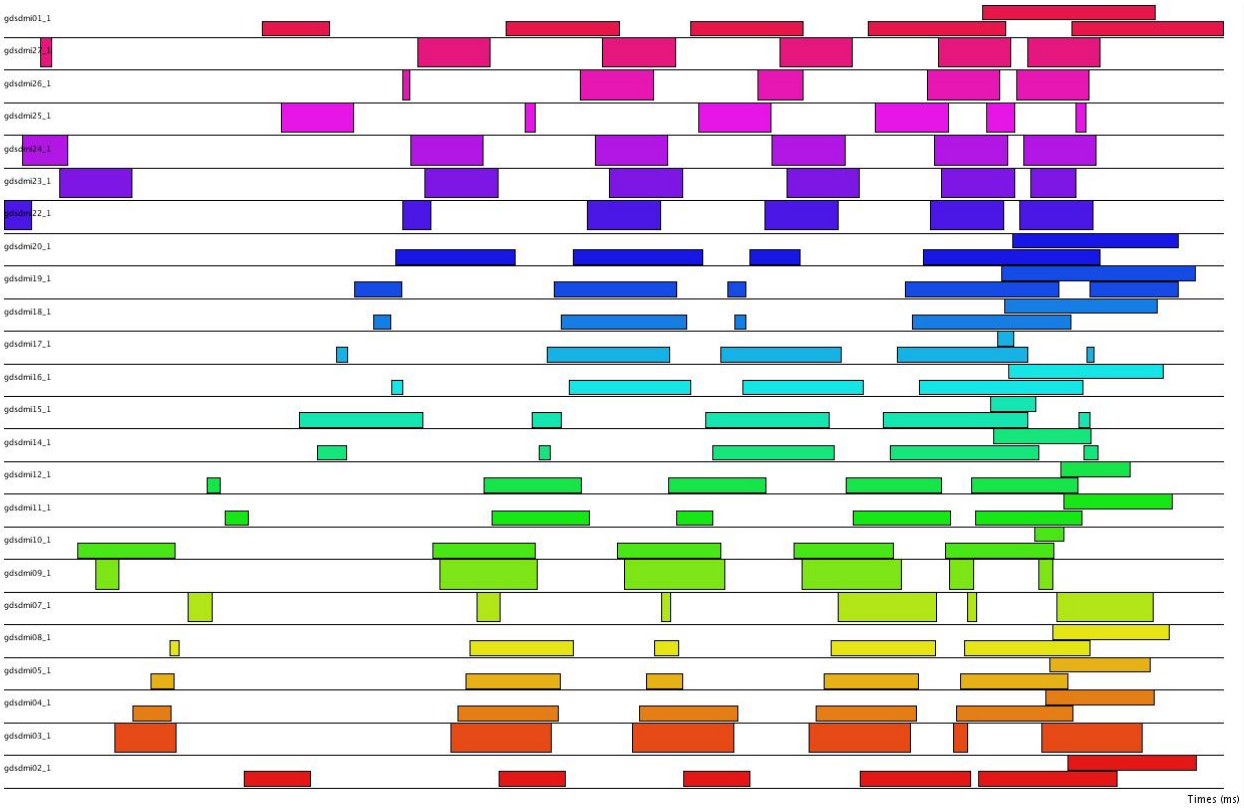

Figure 5.11: The Gantt view of deployed requests on different SeDs for the test 3, round robin scheduling

Conclusion

The load average can be used for particular scheduling with high time intervals. But it is rarely useful for scheduling with high reactivity needs. Therefore we should use the free CPU metric.

5.6 Future works

Even if the new tool CoRI works fine, some improvements are necessary for the future.

- **Non-intrusiveness and low latency**
  - Estimation vectors are always instantiations of specific requests, so they are created for the one request and they will die if the request is scheduled. However the measurements are neither stored persistently for other services nor accessible for other requests. For this reason, a new vector with new measures must be performed for each request; nonetheless it could be possible to reuse the information. The latency and the intrusiveness could

be reduced enormous if a cache mechanism would be installed. Instead of accessing the measure, an old measurement would be provided. At this moment, it would be necessary to integrate a time stamp or a timer in order to refresh the measurement periodically. And this work would bring us back to the sensor and monitor mechanisms.

  - The read and write measures for devices takes too long and uses the full CPU capacity. For further versions of CoRI-Easy, these measure functions should be enhanced.

- **High availability**

  - Other factors can influence the performance of machines, for example the network. That is why new performance measures should be provided by CoRI-Easy.
  - Every operating system provides different basic functions to get the performance information. If the measurements are perceived by operating system functions (fourth level), the function set for this measure must be updated to provide this measure for every operating system.

- We need measures with higher **accuracy**. For example the load average is not sufficiently expressive for handling with different priority level. Which priority level do the processes that occupy the CPU have? Can the new process access directly to CPU resources due to a higher priority level? Another example is the available memory. As the memory is handled differently by the operating systems, it is sometimes difficult to measure the real available memory. Swap technologies and policies are sometimes very different and can falsify the measurements.

- An easy way to provide all these measurements is to use new collectors. Integrating other external tools like Ganglia [MASSIE and al., 04] or JAMM [TIERNEY and al., 00] to the CoRI Manager can provide lower intrusiveness and latency and higher accuracy and availability.

Despite all these problems, CoRI is an important step towards a fully management service for resource information. It will initially enable the service developer to create user defined schedulers with the necessary resource information, and then enable the DIET developer team to define new default scheduling policies.

Summary

In this section we have seen the different requirements for the new grid resource information service CoRI. We have seen the architecture of CoRI Manager and CoRI-Easy, their architectures and their APIs. For CoRI-Easy we have pointed out the difficulties of a low level service that must support multi platforms, and we have proposed some approaches to resolve them. Hence we have detailed for CoRI-Easy the design solution, the different measure functions, and the arising problems. Finally we have made a future perspective for the CoRI tool.

Conclusion

This dissertation about grid resource information services introduced the main components about the grid, the scheduling, and the different classes of information services. An important part of the time was devoted to this state of the art. The practical part consisted of the comprehension of the tool DIET and its accessory programs (VizDIET, GoDIET,...), the comprehension of the lower level details about DIET's implementation in order to integrate the new CoRI tool. Due to the work associated with this dissertation, DIET possess now a basic information service that allows DIET to be independent of other software like FAST or NWS and that provides some basic metrics useful for basic scheduling. But as we have seen in the Section 5.6, there are still some improvements possible and some problems to resolve. In addition, the DIET developer can now integrate other existing information services like Ganglia or Nagios into the DIET toolkit. This allows more accurate information about the grid elements and thereby better scheduling. Also, the service developer can now use the plug-in scheduler facility in combination with CoRI for building user-defined scheduler policies for his services.

In the future, grid computing will become more and more important, and so the number and the size of grids will likewise becomes more important. That is why grid middleware like DIET will take advantage of their special architecture that supports these growing number of grid components. In the sub-domain of resource information services, it will be important to add new collectors that are especially designed for the grid environment. So we can see the CoRI Manager as an open window that allows the use of a lot of different tools developed around the world, and the CoRI-Easy collector as the first version of a homemade resource monitor.

Bibliography

[ANDREOZZI and al, 03] S. Andreozzi, N. De Bortoli, S. Fantinel, A. Ghiselli, G. Tortone, C. Vistoli, *GridICE: A Monitoring Service for the Grid*, in *Proceedings of the Third Cracow Grid Workshop*, Cracow, Poland, October 27-29, 2003, pp. 220-226.

[ARABE and al, 95] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan,*Dome: Parallel programming in a heterogeneous multiŋuser environment*, in *Technical Report TR CMU-CS-95-137*, Carnegie Mellon University, April 1995.

[ARNOLD and al, 01] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar, *Users' Guide to NetSolve V1.4.*, in *Computer Science Dept. Technical Report CS-01-467*, University of Tennessee, Knoxville, TN, July 2001. `http://www.cs.utk.edu/netsolve/`

[AU and al, 96] P. Au, J. Darlington, M. Ghanem, Y. Guo, H. To, and J. Yang, *Coordinating heterogeneous parallel computation*, *Proceedings of the 1996 EuroŋPar Conference*, 1996.

[BERMAN, 99] F. Berman, *High-performance schedulers*, in *Foster, I. and Kesselman, C. eds. The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, pp. 279-309, 1999.

[BERMAN and al., 97] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov0, *Adaptive Computing on the Grid Using AppLeS*, in *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369-382, 2003.

[BOLZE and al., 06] R. Bolze, Y., E. Caron, P. Kaur Chouhan, P. Combes, S. Dahan, H. Dail, B. Delfabro, P. Frauenkron, G. Hoesch, M. Jan, J.-Y. L'xcellent, C. Pera, C. Pontvieux, A. Su, C. Tedeschi, and A. Vernois, *DIET User's Manual*, Version 2.1, January 2001, Copyright INRIA.

[BOTE-LORENZO and al, 04] M. L. Bote-Lorenzo, Y. A. Dimitriadis, and E. Gómez-Sánchez, *Grid characteristics and uses: a grid definition*, in *PostProc. of the 1st European Across Grids Conference*, Santiago de Compostela, Spain, Lecture Notes in Computer Science, 2970, pp. 291-298, February 2004.

[CAMPBELL and al, 04] J. P. Campbell, R. A. McCloy, S. H. Oppler, C. E. Sager, *A theory of performance*, in *E. Schmitt, W. C. Borman, & Associates (Eds.)*, Personnel selection in organizations, (pp. 35-70), San Francisco: Jossey-Bass, 1993.

[CAPPELLO and al., 05] F. Cappello, S. Djilalia, G. Fedak, T. Heraulta, F. Magnettea, V. Nérib and O. Lodygenskyc, *Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid*, in *Future Generation Computer Systems*, 21(3):417-437, March 2005.

[CARON and al, 05] E. Caron and F. Deprez, *DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid*, In *Research Report No 2005-23*, June 2005.

[CARON and al.] E. Caron, F. Desprez, F. Petit, and C. Tedeschi, A Peer-to-Peer Extension of Network-Enabled Server Systems, in *e-Science 2005. First IEEE International Conference on e-Science and Grid Computing*, (pp 430-437), Melbourne, Australia, December 2005.

[CARON and al, 02] E. Caron and F. Suter, *Parallel Extension of a Dynamic Performance Forecasting Tool*, in *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC'02)*, Iasi, Romania, pages 80–93, 2002.

[CHAPIN and al, 98] S. J. Chapin, J. Karpovich, and A. Grimshaw, *Resource management in legion*, in *Technical Report CSy98y09*, University of Virginia, Department of Computer Science, May 1998.

[CSAJKOWSKI and al, 01] K. Czajkowski, C. Kesselman, S. Fitzgerald, I. Foster, *Grid Information Services for Distributed Resource Sharing*, in *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 '01)*, hpdc, p. 0181, 2001.

[DAHAN, 05] S. Dahan, *Mécanismes de recherche de services extensibles pour les environnements de grilles de calcul*, Université de Franche Comté, 2005.

[DAIL and al., 06] H. Dail and F. Desprez, *Experiences with Hierarchical Request Flow Management for Network-Enabled Server Environments*, in *International Journal of High Performance Computing Applications*, Volume 20, Number 1, February 2006

[DEWITT and al., 98] A. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, D. Sutherland, *ReMoS: A Resource Monitoring System for Network-Aware Applications*, Carnegie Mellon School of Computer Science, CMU-CS-97-194, December 1998.

[DINDA and al., 99] P. A. Dinda and D. R. OŠHallaron, *An extensible toolkit for resource prediction in distributed systems*, in *Technical Report CMU-CS-99-138*, CMU, 1999.

[DORST, 06] W. V. Dorst, *BogoMips mini-Howto*, `http://www.clifton.nl/index.html?bogomips.html`, (Version V38, last revised 02/03/2006) (Date of access 28/04/06).

[FOSTER and al, 96] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke, *Software infrastructure for the I-WAY high-performance distributed computing experiment*,in *hpdc* , p. 562, 1996.

[FOSTER and al, 98] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufman Publishers, San Francisco, 1998.

[FOSTER and al, 01] I. Foster, C. Kesselman and Steven Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Lecture Notes in Computer Science, 2001, `citeseer.ist.psu.edu/foster01anatomy.html`.

[FOSTER, 02] I. Foster, *What is the Grid? A Three Point Checklist*, in *GridToday* Vol. 1, No. 6 July 2002, `http://www.gridtoday.com/02/0722/020722.html`.

[GALLANT and al., 92] Gallant, A. Ronald, and George Tauchen , *A Nonparametric Approach to Nonlinear Time Series Analysis: Estimation and Simulation*, in *David Brillinger, Peter Caines, John Geweke, Emanuel Parzen, Murray Rosenblatt, and Murad S. Taqqu eds. New Directions in Time Series Analysis*, Part II. New York: Springer-Verlag, 71-92, 1992

[GAUTAMA and al., 00] H. Gautama and A. J. C. van Gemund *Static performance prediction of data-dependent programs*, in *Proceedings of 2nd International Workshop on Software Performance*, Association for Computing Machinery, New York, N.Y., pp. 216-226, 2000.

[GEHRINF and al., 96] J. Gehrinf and A. Reinfeld, *Mars - a framework for minimizing the job execution time in a metacomputing environment*, in *Proceedings of Future General Computer Systems*, 1996.

[GERNDT and al., 04] M. Gerndt, R. Wismueller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorszki, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef, *Performance Tools for the Grid: State of the Art and Future*, in *Research Report Series*, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen. Shaker Verlag, volume 30, ISBN 3-8322-2413-0, 2004. http://citeseer.ist.psu.edu/gerndt04performance.html

[GIBSON, 59] J. C. Gibson, *The Gibson Mix*, in *Technical Report TR 00.2043*, IBM Systems Development Division, Poughkeepsie, NY, 1970. Widely quoted as a research done in 1959.

GRIDToday, Volume 1, No. 6, July 2002.

[GLOBUS] The Globus Alliance, `www.globus.org`.

[GRID'5000] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard, *Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform*, in *Grid'2005 Workshop*, IEEE/ACM, Seattle, USA, November 2005.

[GRIDRPC] GridRPC Working Group, `https://forge.gridforum.org/projects/gridrpc-wg/`.

[HOWES and al., 99] I. A. Howes, M. C. Smith, and G. S. Good, *Understanding and deploying LDAP directory services*, in *Macmillian Technical Publishing*, ISBN: 1-57870-070-1, 1999.

[JAIN, 91] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*, John Wiley and Sons, Inc., New York, 1991.

[KARONIS and al, 03] N. T. Karonis, B. Toonen, and I. Foster, *MPICH-G2: A grid-enabled implementation of the Message Passing Interface*, J. Parallel Distrib. Comput., 63(5):551-563, May 2003.

[BAILEY and al, 05] D.H. Bailey and A. Snavely, *Performance modeling: Understanding the present and predicting the future*, in *Euro-Par Conference*, August 2005.

[LEE and al, 01] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen and J. Saltz, *A Grid Programming Primer*, `http://www.gridforum.org/7\_APM/APS.htm`, submitted to the Global Grid Forum, August 2001.

[LIANG and al, 05] T. Liang, C. Wu, J. Chang, C. Shieh and P. Fan, *Enabling Software DSM System for Grid Computing*, in *8th International Symposium on Parallel Architectures, Algorithms and Networks*, ispan, pp. 428-435, 2005.

[MASSIE and al., 04] M.L. Massie, B.N. Chun, D.E. Culler, *Ganglia Distributed Monitoring System: Design, Implementation, and Experience*, in *Parallel Computing 30*, pp. 817-840, February 2004.

[MPI] Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/

[NAKADA and al, 99] H. Nakada, M. Sato, and S. Sekiguchi, *Design and Implementations of Ninf: to- wards a Global Computing Infrastructure*, in *Future Generation Computing Systems*, *Metacomputing Issue*, 15(5-6):649-658, 1999. `http://ninf.apgrid.org/papers/papers.shtml`

[NAKADA and al, 02] H. Nakada, S. Matsuoka, K. Seymour, J. Dangarra, C. Lee, H. Casanova *GridRPC: A Remote Procedure Call API for Grid Computing* Grid Computing - GRID 2002, in *Third International Workshop*, Baltimore, MD, USA, Proceedings, Springer , LNCS, Volume 2536, pp. 274-278, November 2002.

[NEWMAN and al, 03] , H.B. Newman, I.C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, *MonALISA : A Distributed Monitoring Service Architecture*, `http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cs/0306096`, 2003.

[OGDEN, 97] R. Ogden, *Essential Wavelets for Statistical Applications and Data Analysis*, in *Birkhauser Boston Inc.*, 1997.

[POSIX, 04] IEEE and The Open Group, *The Single UNIX Specification*, Version 3, 2004 Edition (8 Volumes), `http://www.unix.org/single_unix_specification/`.

[POVRAY] Persistence of Vision Raytracer, http://www.povray.org/

[QUINSON, 02] M. Quinson, *Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment*, in *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems* (PMEO-PDS'02), April 2002.

[SEYMOUR and al, 04] K. Seymour, C. Lee, F. Desprez, H. Nakada and Y. Tanaka, *The End-User and Middleware APIs for GridRPC*, in *Workshop on Grid Application Programming Interfaces*, In conjunction with GGF12, Brussels, Belgium, September 2004.

[SONNENTAG and al, 02] , S. Sonnentag and M. Frese, *Performance Concepts and Performance Theory*, in *Psychological Management of Individual Performance*, S. Sonnentag (ed.), Wiley, Chichester, pp. 3-25, 2002.

[SU, 05] A. Su, *Design and Implemention of a Plug-in Scheduler for DIET*, slides, May 2005.

[TIERNEY and al., 00] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, M. Thompson, *A Monitoring Sensor Management System for Grid Environments*, in *Proceedings of the IEEE High Performance Distributed Computing conference* (HPDC-9), LBNL-45260, August 2000.

[TIERNEY and al., 02] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, R. Wolski, *A Grid Monitoring Architecture*, in *Global Grid Forum*, GWDPerf-16Ű3, August 2002, `http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-3.pdf`.

[WATSON and al., 02] WA. Watson III , I. Bird, J. Chen, B. Hess, A. Kowalski and Y. Chen, *A Web Services Data Analysis Grid*, in *Concurrency and Computation: Practice and Experience*, Vol. 14, Grid Computing environments Special Issue 13-15, pages 1303-1312, 2002

[WOLSKI and al, 99] R.Wolski, N. T. Spring and J. Hayes, *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, in *Future Generation Computing Systems*, in *Metacomputing Issue*, 15(5-6):757-768, October 1999.

[XUEHAI and al., 03] X. Zhang, J. Freschl, and J. M. Schopf, *A performance study of monitoring and information services for distributed systems*, in *Proceedings of the IEEE Twelfth International Symposium on High-Performance Distributed Computing*, (HPDC-12), 2003.